CORE JAVA

Disclaimer

The copyright of the content used in the courseware will remain with Principle Company.

TABLE OF CONTENTS

01

Introduction

1.1 What is Java?

The term Java refers to more than just a computer language like C or Pascal. Java encompasses several distinct components:

A high-level language – Java is an object-oriented language whose source code at a glance looks very similar to C and C++ but is unique in many ways.

Java bytecode – A compiler transforms the Java language source code to files of binary instructions and data called bytecode that run in the Java Virtual Machine.

Java Virtual Machine (JVM) – A JVM program takes bytecode as input and interprets the instructions just as if it were a physical processor executing machine code. Sun Microsystems owns the Java trademark (see the next section on the history of Java) and provides a set of programming tools and class libraries in bundles called Java Software Development Kits (SDKs). The tools include javac, which compiles Java source code into bytecode, and java, the executable program that creates a JVM that executes the bytecode. Sun provides SDKs for Windows, Linux, and Solaris. Other vendors provide SDKs for their own platforms (IBM AIX and Apple Mac OS X, for example). Sun also provides a runtime bundle with just the JVM and a few tools for users who want to run Java programs on their machines but have no intention of creating Java programs. This runtime bundle is called the Java Runtime Environment (JRE).

In hope of making Java a widely used standard, Sun placed minimal restrictions on Java and gave substantial control of the development of the language over to a broadly based Java community organization (see Section 1.4 “Java: open or closed?”). So as long as other implementations obey the official Java specifications, any or all of the Java components can be replaced by non-Sun components. For example, just as compilers for different languages can create machine code for the same processor, there are programs for compiling source code written in other languages, such as Pascal and C, into Java bytecode. There are even Java bytecode assembler programs. Many JVMs have been written by independent sources.

Java might be said more accurately to refer to a set of programming and computing specifications. However, in this book and the Web Course, unless otherwise indicated, we follow the common use of the term Java to refer to the high-level language that follows the official specifications and the virtual machine platform on which the compiled language runs. The usage is normally clear from the context. Finally, many people know Java only from the applets that run in their web browsers. Java programs, however, can also run as standalone programs just like any other language. Such standalone programs are referred to as "applications" to distinguish them from applets.

1.2 History of Java

During 1990, James Gosling, Bill Joy and others at Sun Microsystems began developing a language called Oak. They primarily intended it as a language for microprocessors embedded in consumer devices such as cable set-top boxes, VCRs, and handheld computers (now known as personal data assistants or PDAs). To serve these goals, Oak needed the following features:

platform independence, since it must run on devices from multiple manufacturers **extreme reliability** (can' t expect consumers to reboot their VCRs!) **compactness**, since embedded processors typically have limited memory

They also wanted a next-generation language that built on the strengths and avoided the weaknesses of earlier languages. Such features would help the new language provide more rapid software development and faster debugging.

By 1993 the interactiveTVand PDAmarkets had failed to take off, but internet and web activity began its upward zoom. So Sun shifted the target market to internet applications and changed the name of the project to Java. The portability of Java made it ideal for the Web, and in 1995 Sun's HotJava browser appeared. Written in Java in only a few months, it illustrated the power of applets – programs that run within a browser – and the ability of Java to accelerate program development. Riding atop the tidal wave of interest and publicity in the Internet, Java quickly gained widespread recognition (some would say hype), and expectations grew for it to become the dominant software for browsers and perhaps even for desktop programs. However, the early versions of Java did not possess the breadth and depth of capabilities needed for desktop applications. For example, the graphics in Java 1.0 appeared crude and clumsy compared with the graphics features available in software written in C and other languages and targeted at a single operating system.

Applets did in fact become popular and remain a common component of web page design. However, they do not dominate interactive or multimedia displays in the browser as expected. Many other "plug-in" programs also run within the browser environment.

Though Java's capabilities grew enormously with the release of several expanded versions (see Section 1.3), Java has not yet found wide success in desktop client applications. Instead Java gained widespread acceptance at the two opposite ends of the platform spectrum: large business systems and very small systems.

Java is used extensively in business to develop enterprise, or middleware, applications such as on-line stores, transactions processing, dynamic web page generation, and database interactions. Java has also returned to its Oak roots and become very common on small platforms such as smart cards, cell phones, and PDAs. For example, as of mid-2004 there are over 350 different Java-enabled mobile phone handsets available across the world, and over 600 million Java Cards have been distributed.

1.3 Versions of Java

Since its introduction, Sun has released new versions of the Java core language with significant enhancements about every two years or so. Until recently, Sun denoted the versions with a 1.x number, where x reached up to 4. (Less drastic releases with bug fixes were indicated with a third number as in 1.4.2.) The next version, however, will be called Java 5.0. Furthermore, Sun has split its development kits into so-called editions, each aimed towards a platform with different capabilities. Here we try to clarify all of this.

1.3.1 Standard Edition

Below is a time line for the different versions of the Standard Edition (SE) of Java, which offers the core language libraries (called packages in Java) and is aimed at desktop platforms.We include a sampling of the new features that came with each release.

1995 - Version 1.0. The Java Development Kit (JDK) included:
8 packages with 212 classes.
Netscape 2.0–4.0 included Java 1.0.
Microsoft and other companies licensed Java.

1997 - Version 1.1
23 packages, 504 classes.
Improvements included better event handling, inner classes, improved VM. Microsoft developed its own 1.1-compatible Java Virtual Machine for the Internet Explorer.
Many browsers in use are still compatible only with 1.1. Swing packages with greatly improved graphics became available during this time but were not included with the core language.

1999 - Version 1.2. Sun began referring to the 1.2 and above versions as the Java 2 Platform.
The Software Development Kit (SDK) included: 59 packages, 1520 classes.

Java Foundation Classes (JFC), based on Swing, for improved graphics and user interfaces, now included with the core language. Collections Framework API included support for various lists, sets, and hash maps.

2000 - Version 1.3:
76 packages, 1842 classes.
Performance enhancements including the Hotspot virtual machine.

2002 - Version 1.4:
135 packages, 2991 classes.

Improved IO, XML support, etc.

2004 ‑ Version 5.0 (previously known as 1.5). This version was available only in beta release as this book went to press. See Section 1.9 for an overview of what is one of the most extensive updates of Java since version 1.0. It includes a number of tools and additions to the language to enhance program development, such as:

faster startup and smaller memory footprint
metadata
formatted output
generics
improved multithreading features
165 packages, over 3000 classes

During the early years, versions for Windows platforms and Solaris (Sun's version of Unix) typically became available before Linux and the Apple Macintosh. Over the last few years, Sun has fully supported Linux and has released Linux, Solaris, and Windows versions of Java simultaneously. Apple Computer releases its own version for the Mac OS X operating system, typically a few months after the official Sun release of a new version. In addition, in the year 1999, Sun split off two separate editions of Java 2 software under the general categories of Micro and Enterprise editions, which we discuss next.

1.3.2 Other editions

Embedded processor systems, such as cell phones and PDAs, typically offer very limited resources as compared to desktop PCs. This means small amounts of RAM and very little disk space or non-volatile memory. It also usually means small, low-resolution displays, if any at all. So Sun offers slimmed-down versions of Java for such applications. Until recently this involved three separate bundles of Java 1.1-based packages, organized according to the size of the platform. Java Card is intended for extremely limited Java for systems with only 16 KB non-volatile memory and 512 bytes volatile. The Embedded Java and Personal Java bundles are intended for systems with memory resources in the 512 KB and 2 MB ranges, respectively. To provide a more unified structure to programming for small platforms, Sun has replaced Embedded Java and Personal Java (but not JavaCard) with the Java 2 Micro Edition (J2ME). The developer chooses from different subsets of the packages to suit the capacity of a given system. At the other extreme are high-end platforms, often involving multiple processors, that carry out large-scale computing tasks such as online stores, interactions with massive databases, etc. With the Java 2 Platform came a separate set of libraries called the Java 2 Enterprise Edition (J2EE) with enhanced resources targeted at these types of applications. Built around the same core as Standard Edition packages, it provides an additional array of tools for building these so called middleware products.

1.3.3 Naming conventions

We note that the naming and version numbering scheme in Java can be rather confusing. As we see from the time line above, the original Java release included the Java Development Kit (JDK) and was referred to as either Java 1.0 or JDK 1.0. Then came JDK 1.1 with a number of significant changes. The name Java 2 first appeared with what would have been JDK 1.2. At that time the JDK moniker was dropped in favor of SDK (for Software Development Kit). Thus the official name of the first Java 2 development kit was something like Java 2 Platform Standard Edition (J2SE) SDK Version 1.2. Versions 1.3 and 1.4 continued the same naming/numbering scheme.

Meanwhile many people continue to use the JDK terminology – thus JDK 1.4 when referring to J2SE SDK Version 1.4. Another common usage is the simpler but less specific Java Version 1.x, or even just Java 1.x to mean J2SE Version 1.x. Both of these usages are imprecise because there is also a Java 2 Enterprise Edition (J2EE) Version 1.4. To make it clear what you mean, you should probably either use J2SE or J2EE rather than just Java when mentioning a version number unless the meaning is clear from context. This book is not about J2EE, though we do touch on Java Servlet technology in Chapters 14 and 21 and on web services in general in Chapter 21. Since we never need to refer to the Enterprise Edition, we use the terms Java 1.x, SDK 1.x, and J2SE 1.x interchangeably.

By the time this book is in your hands, Sun will have released the Java 2 Standard Edition Version 5.0. The version number 5.0 replaces what would have been version number 1.5. Undoubtedly many people will continue to use the 1.5 terminology. In fact, the Beta 2 release of J2SE 5.0 (the latest available at the time of this writing) continues to use the value 1.5 in some places. You may also come across the code name Tiger for the 5.0 release; however, we expect that usage to fade away just like previous code names Kestrel and Merlin have all but disappeared from the scene. This book uses the notation J2SE 5.0 or Release 5.0 or Version 5.0 or Java 5.0 or sometimes just 5.0 when referring to this very significant new release of Java.

We provide a brief overview of Java 5.0 in Section 1.9 and examine a number of 5.0 topics in some detail throughout the book.

1.4 Java – open or closed?

Java is not a true open language but not quite a proprietary one either. All the core language components – compiler, virtual machines, core language class packages, and many other tools – are free from Sun. Furthermore, Sun makes detailed specifications and source code openly available for the core language. Another company can legally create a so-called clean room compiler and/or a Java Virtual Machine as long as it follows the detailed publicly available specifications and agrees to the trademark and licensing terms. Microsoft, for example, created its own version 1.1 JVM for the Internet Explorer browser. See the Web Course for a listing of other independent Java compilers and virtual machines. Sun, other companies, and independent programmers participate in the Java Community Process (JCP) organization whose charter is “to develop and revise Java technology specifications, reference implementations, and technology compatibility kits.” Proposals for newAPIs, classes, and other changes to the language now follow a formal process in the JCP to achieve acceptance.

Sun, however, does assert final say on the specifications and maintains various restrictions and trademarks (such as the Java name). For example, Microsoft’s JVM differed in some significant details from the specifications and Sun filed a lawsuit (later settled out of court) that claimed Microsoft attempted to weaken Java’s “Write Once, Run Anywhere” capabilities.

1.5 Java features and benefits

Before we examine how Java can benefit technical applications, we look at the features that make Java a powerful and popular general programming language.

These features include:

Compiler/interpreter combination

• Code is compiled to bytecode, which is interpreted by a Java Virtual Machine (JVM). This provides portability to any base operating system platform for which a virtual machine has been written.

- The two-step procedure of compilation and interpretation allows for extensive code checking and improved security.

Object-oriented

• Object-oriented programming (OOP) throughout - no coding outside of class definitions.
• The bytecode retains an object structure.
• An extensive class library available in the core language packages.

Automatic memory management

• A garbage collector in the JVM takes care of allocating and reclaiming memory.

•

Several drawbacks of C and C++ eliminated

· No accessible memory pointers.
• No preprocessor.
• Array limits automatically checked.

Robust

• Exception handling built-in, strong type checking (that is, all variables must be assigned an explicit data type), local variables must be initialized.

Platform independence

- The bytecode runs on any platform with a compatible JVM.
- The "Write Once Run Anywhere" ideal has not been achieved (tuning for different platforms usually required), but is closer than with other languages.

Security

- The elimination of direct memory pointers and automatic array limit checking prevents rogue programs from reaching into sections of memory where they shouldn't.
- Untrusted programs are restricted to run inside the virtual machine sandbox. Access to the platform can be strictly controlled by a Security Manager.
- Code is checked for pathologies by a class loader and a bytecode verifier.
- Core language includes many security related tools, classes,

etc.

Dynamic binding

- Classes, methods, and variables are linked at runtime.

Good performance

• Interpretation of bytecodes slowed performance in early versions, but advanced virtual machines with adaptive and just-in-time compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs.

Threading

• Lightweight processes, called threads, can easily be spun off to perform multiprocessing.

Built-in networking

• Java was designed with networking in mind. The core language packages come with many classes to program Internet communications.

· The Enterprise Edition provides an extensive set of tools for building middleware systems for advanced network applications.

These features provide a number of benefits compared to program development in other languages. For example, C/C++ programs are beset by bugs resulting from direct memory pointers, which are eliminated in Java. Similarly, the array limit checking prevents another common source of bugs. The garbage collector relieves the programmer of the big job of memory management. It's often said that these features lead to a significant speedup in program development and debugging compared to working with C/C++.

1.5.1 Java features and benefits for technical programming

The above features benefit all types of programming. For science and engineering specifically, Java provides a number of advantages:

Platform independence – Engineers and scientists, particularly experimentalists, probably use more types of computers and operating systems that any other group. Code that can run on different machines without rewrites and recompilation saves time and effort.

Object-oriented – Besides the usual benefits from OOP, scientific programming can often benefit from thinking in terms of objects. For example, atomic particles in a scattering simulation are naturally self-contained objects.

Threading – Multiprocessing is very useful for many scientific tasks such as simulations of phenomena where many processes occur simultaneously. This can be quite useful in the conceptual design of a program even when it will run on a single-processor machine. However, Java Virtual Machines on multiprocessor platforms also can distribute threads to the different processors to obtain true parallel performance.

Simulation tools – The extensive graphics resources and multithreading in the core Java language provide for depicting and animating engineering and scientific devices and phenomena.

Networking – Java comes with many networking capabilities that allow one to build distributed systems. Such capabilities can be applied, for example, to data collection from remote sensors.

Interfacing and enhancing legacy code – Java' s strong graphics and networking capabilities can be applied to existing C and Fortran programs. A Java graphical user interface (GUI) can bring enhanced ease of use to a Fortran or C program, which then acts as a computational engine behind the GUI.

1.5.2 Java shortcomings for technical programming

Several features of Java that make it a powerful and highly secure language, such as array limit checking and the absence of direct memory pointers, can also slow it down, especially for large-scale intensive mathematical calculations. Furthermore, the interpretation of bytecode that makes Java programs so easily portable can cause a big reduction in performance as compared to running a program in local machine code. This was a particular problem in the early days of Java, and its slow performance led many who experimented with Java to drop it.

Fortunately, more sophisticated JVMs have greatly improved Java performance. So called Just-in-Time compilers, for example, convert bytecode to local machine code on the fly. This is especially effective for repetitive sections of code. During the first pass through a loop the code is interpreted and converted to native code so that in subsequent passes the code will run at full native speeds.

Another approach involves adaptive interpretation, such as in Sun's Hotspot, in which the JVM dynamically determines where performance in a program is slow and then optimizes that section in local machine code. Such an approach can actually lead to faster performance than C/C++ programs in some cases.

Here are some other problems in applying Java to technical programming:

No rectangular arrays – Java 2D arrays are actually 1D arrays of references to other 1D arrays. For example, a 4×4 sized 2D array in Java behaves internally like a single 1D array whose elements point to four other 1D arrays:

A[0] ==> B[0] B[1] B[2] B[3] B[4]
A[1] ==> C[0] C[1] C[2] C[3] C[4]
A[2] ==> D[0] D[1] D[2] D[3] D[4]
A[3] ==> E[0] E[1] E[2] E[3] E[4]

The B, C, D, and E arrays could be in widely separated locations in memory. This differs from Fortran or C in which 2D array elements are contiguous in memory as in this 4×4 array:

A(0,0) A(0,1) A(0,2) A(0,3) A(0,4)
A(1,0) A(1,1) A(1,2) A(1,3) A(1,4)
A(2,0) A(2,1) A(2,2) A(2,3) A(2,4)
A(3,0) A(3,1) A(3,2) A(3,3) A(3,4)

Therefore, with the Java arrays, moving from one element to the next requires extra memory operations as compared to simply incrementing a pointer as in C/C++. This slows the processing when the calculations require multiple operations on large arrays.

No complex primitive type – Numerical and scientific calculations often require imaginary numbers but Java does not include a complex primitive data type. You can easily create a complex number class, but the processing is slower than if a primitive type had been available.

No operator overloading – For coding of mathematical equations and algorithms it would be very convenient to have the option of redefining (or "overloading") the definition of operators such as "+" and "-". In C++, for example, the addition of two complex number objects could use c1 + c2, where you define the + operator to properly add such objects. In Java, you must use method invocations instead. This lengthens the code.

1.6 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Pascal, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

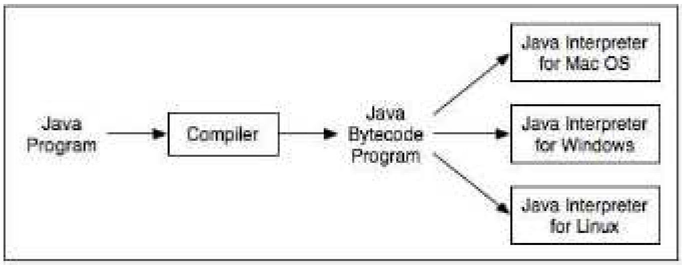
There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an interpreter, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: they can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, there is a program called "Virtual PC" that runs on Macintosh computers. Virtual PC is an interpreter that executes machine-language programs written for IBM-PC-clone computers. If you run Virtual PC on your Macintosh, you can run any PC program, including programs written for Windows. (Unfortunately, a PC program will run much more slowly than it would on an actual IBM clone. The problem is that Virtual PC executes several Macintosh machine-language instructions for each PC machine-language instruction in the program it is interpreting. Compiled programs are inherently faster than interpreted programs.)

The designers of Java chose to use a combination of compilation and interpretation. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn't really exist. This so-called "virtual" computer is known as the Java virtual machine. The machine language for the Java virtual machine is called Java bytecode. There is no reason why Java bytecode could not be used as the machine language of a real computer, rather than a virtual computer.

However, one of the main selling points of Java is that it can actually be used on any computer. All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the Java virtual machine in the same way that Virtual PC simulates a PC computer.

Of course, a different Jave bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program. And the same Java bytecode program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.


Java Program
Compiler
Java Bytecode Program
Java Interpreter for Mac OS
Java Interpreter for Windows
Java Interpreter for Linux

Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are many reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program. A Java bytecode interpreter, on the other hand, is a fairly small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, many Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

I should note that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages could be compiled into Java bytecode. However, it is the combination of Java and Java bytecode that is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

02

Language Fundamentals

2.1 Basic Language Elements

Like any other programming language, the Java programming language is defined by grammar rules that specify how syntactically legal constructs can be formed the language elements, and by a semantic definition that specifies the meaning of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called lexical tokens (or just tokens for short) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

Identifiers

A name in a program is called an identifier. Identifiers can be used to denote classes, methods, variables, and labels.

In Java an identifier is composed of a sequence of characters, where each character can be either a letter, a digit, a connecting punctuation (such as underscore _), or any currency symbol (such as $, •, •, or •). However, the first character in an identifier cannot be a digit. Since Java programs are written in the Unicode character set (see p. 23), the definitions of letter and digit are interpreted according to this character set. Identifiers in Java are case sensitive, for example, price and Price are two different identifiers.

Examples of Legal Identifiers:
number, Number, sum_$, bingo, $$_100, mål, grüß

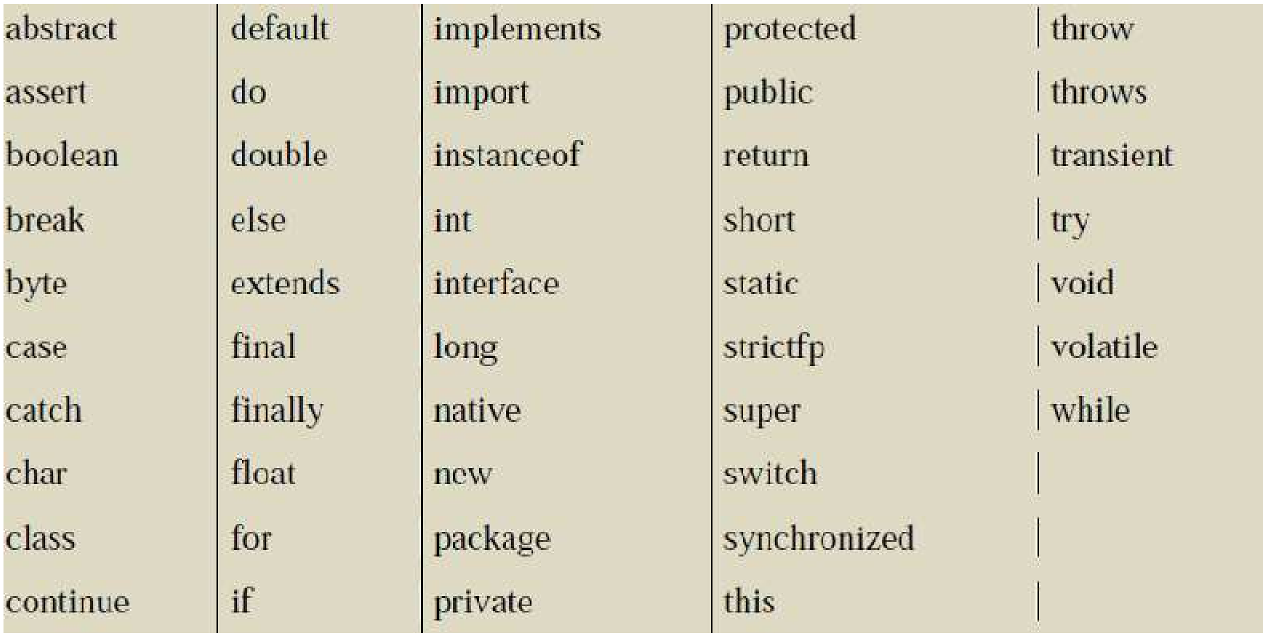
Examples of Illegal Identifiers:
48chevy, all@hands, grand-sum

The name 48chevy is not a legal identifier as it starts with a digit. The character @ is not a legal character in an identifier. It is also not a legal operator so that all@hands cannot not be interpreted as a legal expression with two operands. The character - is also not a legal character in an identifier. However, it is a legal operator so grand-sum could be interpreted as a legal expression with two operands.

Keywords

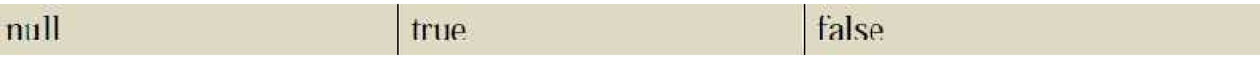
Keywords are reserved identifiers that are predefined in the language and cannot be used to denote other entities. All the keywords are in lowercase, and incorrect usage results in compilation errors.

Keywords currently defined in the language are listed in Table 2.1. In addition, three identifiers are reserved as predefined literals in the language: the null reference and the Boolean literals true and false (see Table 2.2). Keywords currently reserved, but not in use, are listed in Table 2.3. All these reserved words cannot be used as identifiers. The index contains references to relevant sections where currently defined keywords are explained.



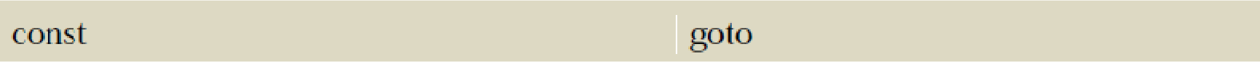
abstract	default	implements	protected	throw
assert	do	import	public	throws
boolean	double	instanceof	return	transient
break	else	int	short	try
byte	extends	interface	static	void
case	final	long	strictfp	volatile
catch	finally	native	super	while
char	float	new	switch	
class	for	package	synchronized	
continue	if	private	this	

Table 2.1: Java Language Keywords



null	true	false

Table 2.2: Reserved Literals in Java

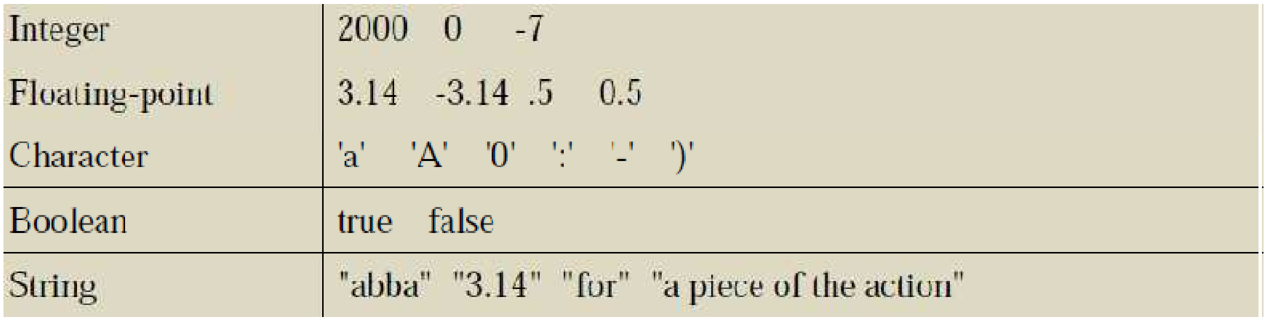


const	goto

Table 2.3:Reserved Keywords not currently in use

Literals

A literal denotes a constant value, that is, the value a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean or string values. In addition, there is the literal null that represents the null reference.



Integer	2000 0 -7
Floating-point	3.14 -3.14 .5 0.5
Character	'a' 'A' '0' ':' '-' ')'
Boolean	true false
String	"abba" "3.14" "for" "a piece of the action"

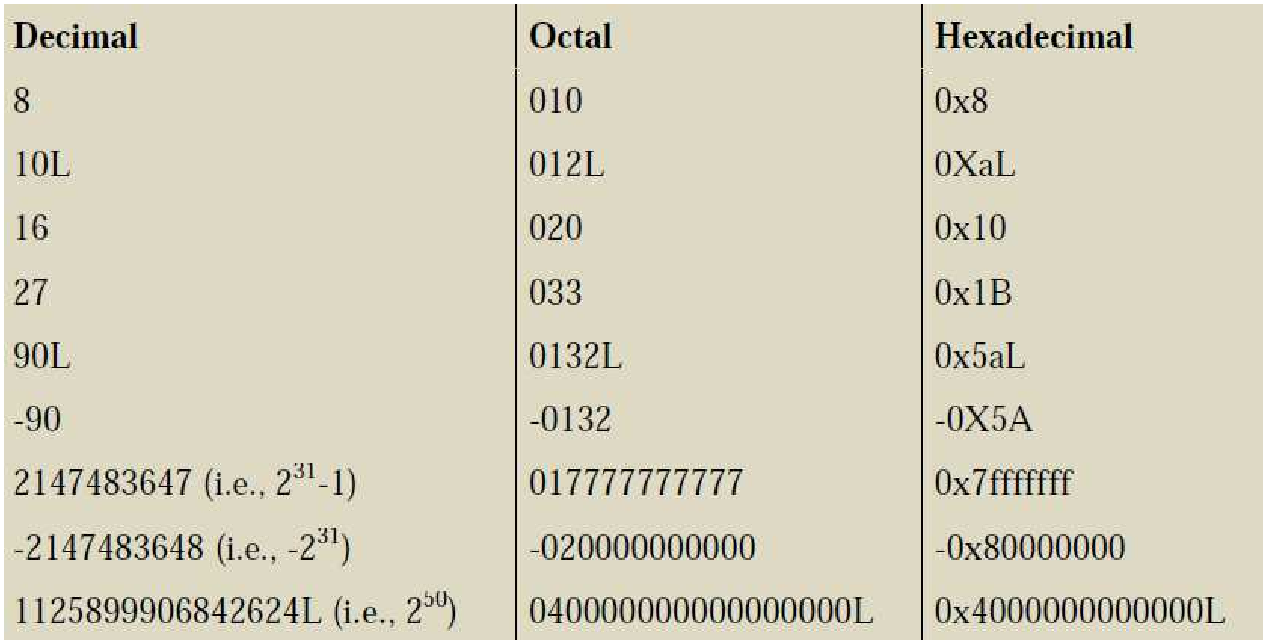
Table 2.4: Examples of Literals

Integer Literals

Integer data types are comprised of the following primitive data types: int, long, byte, and short.

The default data type of an integer literal is always int, but it can be specified as long by appending the suffix L (or l) to the integer value. Without the suffix, the long literals 2000L and 0l will be interpreted as int literals. There is no direct way to specify a short or a byte literal.

In addition to the decimal number system, integer literals can also be specified in octal (base 8) and hexadecimal (base 16) number systems. Octal and hexadecimal numbers are specified with 0 and 0x (or 0X) prefix respectively. Examples of decimal, octal and hexadecimal literals are shown in Table 2.5. Note that the leading 0 (zero) digit is not the uppercase letter O. The hexadecimal digits from a to f can also be specified with the corresponding uppercase forms (A to F). Negative integers (e.g. -90) can be specified by prefixing the minus sign (-) to the magnitude of the integer regardless of number system (e.g., -0132 or -0X5A). Java does not support literals in binary notation.



Decimal	Octal	Hexadecimal
8	010	0x8
10L	012L	0XaL
16	020	0x10
27	033	0x1B
90L	0132L	0x5aL
-90	-0132	-0X5A
2147483647 (i.e., 2^{31}-1)	017777777777	0x7fffffff
-2147483648 (i.e., -2^{31})	-020000000000	-0x80000000
1125899906842624L (i.e., 2^{50})	04000000000000000L	0x400000000000L

Table 2.5: Examples of Decimal, Octal and Hexadecimal Literals

Floating-point Literals

Floating-point data types come in two flavors: float or double. The default data type of a floating-point literal is double, but it can be explicitly designated by appending the suffix D (or d) to the value. A floating-point literal can also be specified to be a float by appending the suffix F (or f). Floating-point literals can also be specified in scientific notation, where E (or e) stands for Exponent. For example, the double literal 194.9E-2 in scientific notation is -2interpreted as 194.9*10 (i.e., 1.949).

Examples of double Literals

```
0.0       0.0d       0D
0.49      .49       .49D
49.0      49.        49D
4.9E+1   4.9E+1D   4.9e1d   4900e-2  .49E2
```

Examples of float Literals

```
0.0F     0f
0.49F    .49F
49.0F    49.F     49F
4.9E+1F  4900e-2f  .49E2F
```

Note that the decimal point and the exponent are optional and that at least one digit must be specified.

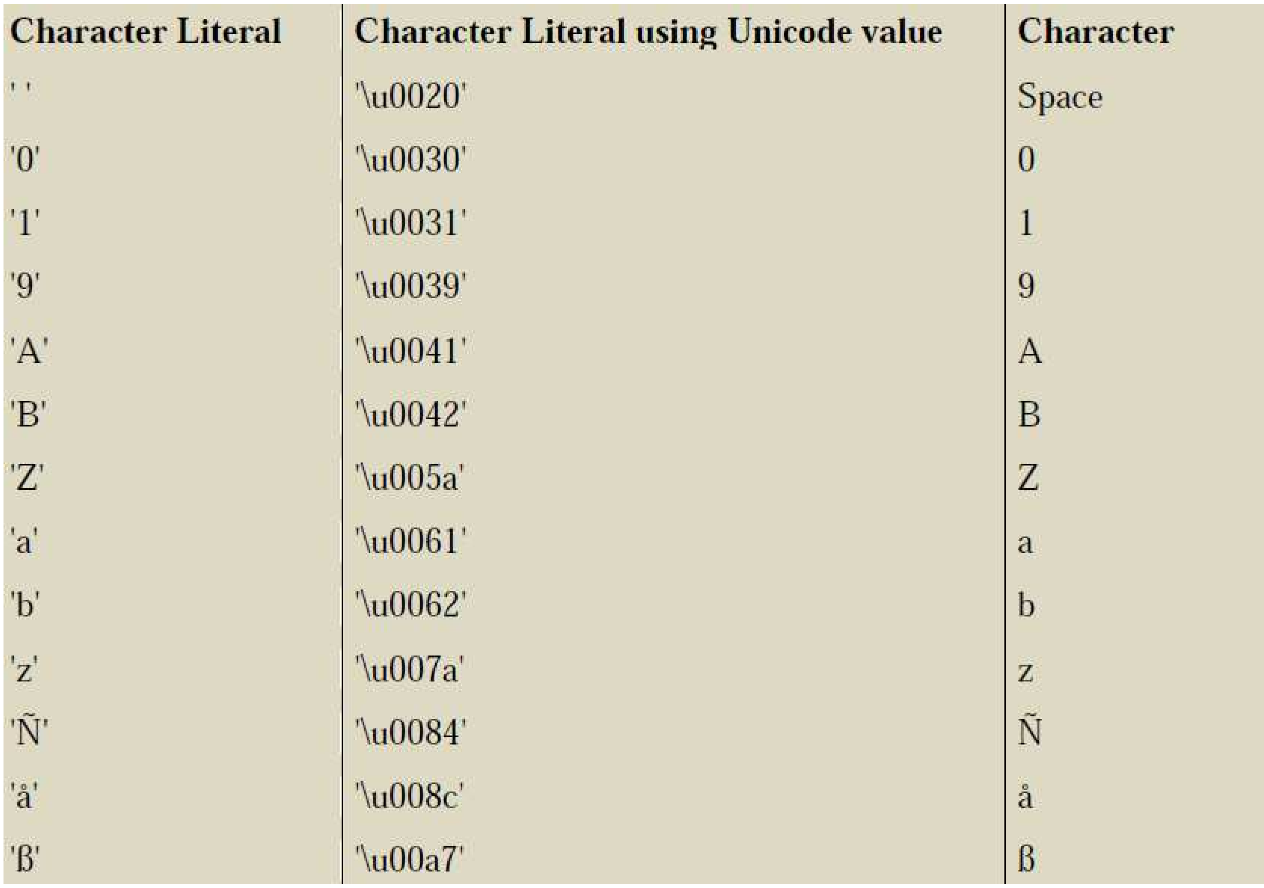
Boolean Literals

The primitive data type boolean represents the truth-values true or false that are denoted by the reserved literals true or false, respectively.

Character Literals

A character literal is quoted in single-quotes ('). All character literals have the primitive data type char.

Characters in Java are represented by the 16-bit Unicode character set, which subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. In Table 2.6, note that digits (0 to 9), upper-case letters (A to Z), and lower-case letters (a to z) have contiguous Unicode values. Any Unicode character can be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix \u.

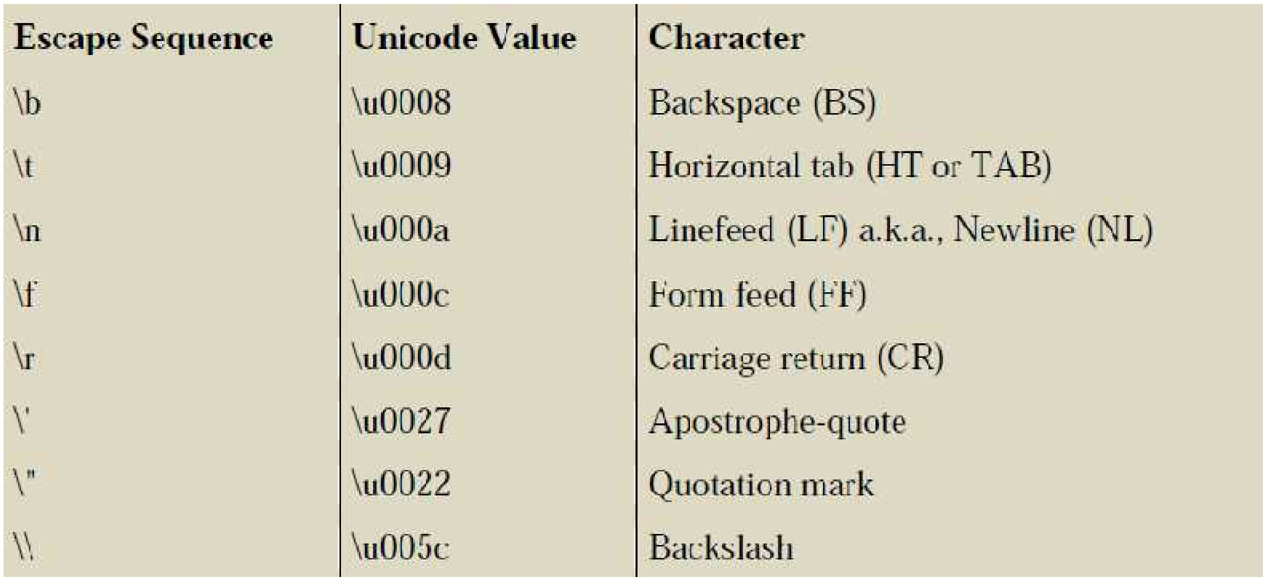


Character Literal	Character Literal using Unicode value	Character
' '	'\u0020'	Space
'0'	'\u0030'	0
'1'	'\u0031'	1
'9'	'\u0039'	9
'A'	'\u0041'	A
'B'	'\u0042'	B
'Z'	'\u005a'	Z
'a'	'\u0061'	a
'b'	'\u0062'	b
'z'	'\u007a'	z
'Ñ'	'\u0084'	Ñ
'å'	'\u008c'	å
'ß'	'\u00a7'	ß

Table 2.6: Examples of Unicode values

Escape Sequences

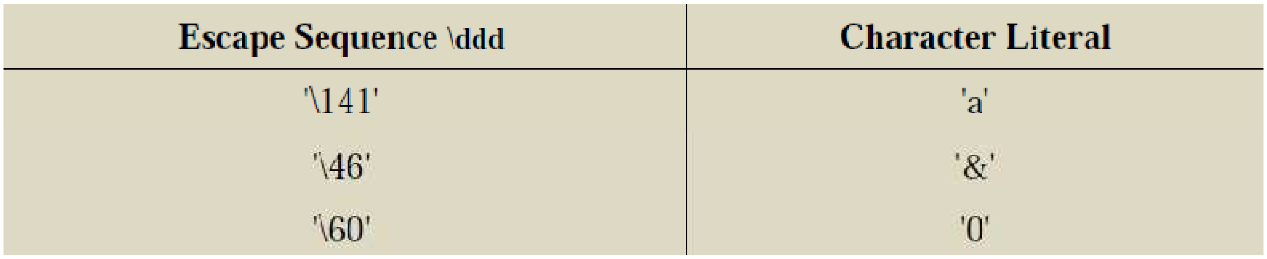
Certain escape sequences define special character values as shown in Table 2.7. These escape sequences can be single-quoted to define character literals. For example, the character literals '\t' and '\u0009' are equivalent. However, the character literals '\u000a' and '\u000d' should not be used to represent newline and carriage return in the source code. These values are interpreted as line-terminator characters by the compiler, and will cause compile time errors. One should use the escape sequences '\n' and '\r', respectively, for correct interpretation of these characters in the source code.



Escape Sequence	Unicode Value	Character
\b	\u0008	Backspace (BS)
\t	\u0009	Horizontal tab (HT or TAB)
\n	\u000a	Linefeed (LF) a.k.a., Newline (NL)
\f	\u000c	Form feed (FF)
\r	\u000d	Carriage return (CR)
\'	\u0027	Apostrophe-quote
\"	\u0022	Quotation mark
\\	\u005c	Backslash

Table 2.7: Escape Sequences

We can also use the escape sequence \ddd to specify a character literal by octal value, where each digit d can be any octal digit (0–7), as shown in the following table. The number of digits must be three or fewer, and the octal value cannot exceed \377, that is, only the first 256 characters can be specified with this notation.



Escape Sequence \ddd	Character Literal
'\141'	'a'
'\46'	'&'
'\60'	'0'

String Literals

A string literal is a sequence of characters, which must be quoted in quotation marks and which must occur on a single line. All string literal are objects of the class String .

Escape sequences as well as Unicode values can appear in string literals:

```
"Here comes a tab.\t And here comes another one\u0009!                    (1)
"What's on the menu?"                                                     (2)
"\"String literals are double-quoted.\""                                  (3)
"Left!\nRight!"                                                           (4)
```

In (1), the tab character is specified using the escape sequence and the Unicode value respectively. In (2), the single apostrophe need not be escaped in strings, but it would be if specified as a character literal('\''). In (3), the double apostrophes in the string must be escaped. In

(4), we use the escape sequence \n to insert a newline. Printing these strings would give the following result:

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
Left!
Right!
```

One should also use the string literals "\n" and "\r", respectively, for correct interpretation of the characters "\u000a" and "\u000d" in the source code.

White Spaces

A white space is a sequence of spaces, tabs, form feeds, and line terminator characters in a Java source file. Line terminators can be newline, carriage return, or carriage return-newline sequence.

A Java program is a free-format sequence of characters that is tokenized by the compiler, that is, broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes white space has to be inserted explicitly as separators. For example, the identifier classRoom will be interpreted as a single token, unless white space is inserted to distinguish the keyword class from the identifier Room.

White space aids not only in separating tokens, but also in formatting the program so that it is easy for humans to read. The compiler ignores the white spaces once the tokens are identified.

Comments

A program can be documented by inserting comments at relevant places. These comments are for documentation purposes and are ignored by the compiler.

Java provides three types of comments to document a program:

- A single-line comment: // ... to the end of the line
- A multiple-line comment: /* ... */
- A documentation (Javadoc) comment: /** ... */

Single-line Comment

All characters after the comment-start sequence // through to the end of the line constitute a single-line comment.

// This comment ends at the end of this line. int age;   // From comment-start sequence to the end of the line is a comment.

Multiple-line Comment

A multiple-line comment, as the name suggests, can span several lines. Such a comment starts with /* and ends with */.

```
/*  A comment
   on several
   lines.
*/
```

The comment-start sequences (//, /*, /**) are not treated differently from other characters when occurring within comments, and are thus ignored. This means trying to nest multiple-line comments will result in compile time error:

```
 /*  Formula for alchemy.
   gold = wizard.makeGold(stone);
   /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence /* is ignored. The last occurrence of the sequence */ in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A documentation comment is a special-purpose comment that when placed before class or class member declarations can be extracted and used by the javadoc tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods and field definitions. Groups of special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with /** and ends with */: /**

```
 *  This class implements GenericPlugin
 *  @author Tom
 *  @version 1.0
 */
```

2.2 Primitive Data Types

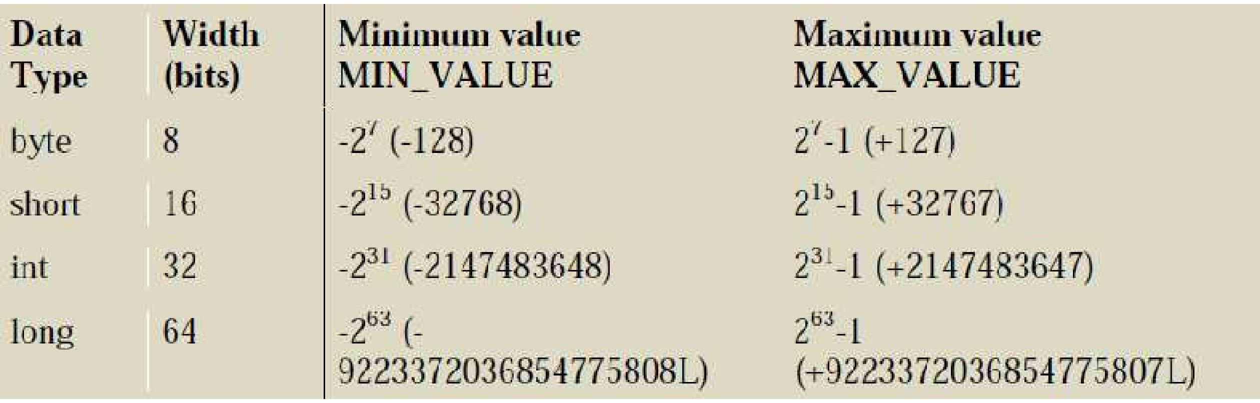
Primitive data types in Java can be divided into three main categories:

- Integral types– represent signed integers (byte, short, int, long) and unsigned character values (char)
- Floating-point types (float, double)– represent fractional signed numbers
- Boolean type (boolean)– represent logical values

Primitive data values are not objects. Each primitive data type defines the range of values in the data type, and operations on these values are defined by special operators in the language.

Each primitive data type also has a corresponding wrapper class that can be used to represent a primitive value as an object.

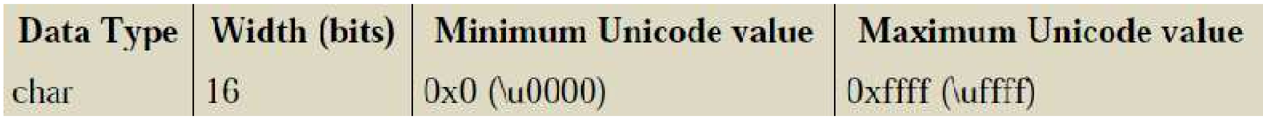
Integer Types



Data Type	Width (bits)	Minimum value MIN_VALUE	Maximum value MAX_VALUE
byte	8	-2^7 (-128)	2^7-1 (+127)
short	16	-2^{15} (-32768)	2^{15}-1 (+32767)
int	32	-2^{31} (-2147483648)	2^{31}-1 (+2147483647)
long	64	-2^{63} (-9223372036854775808L)	2^{63}-1 (+9223372036854775807L)

Their values are signed integers represented by 2's complement.

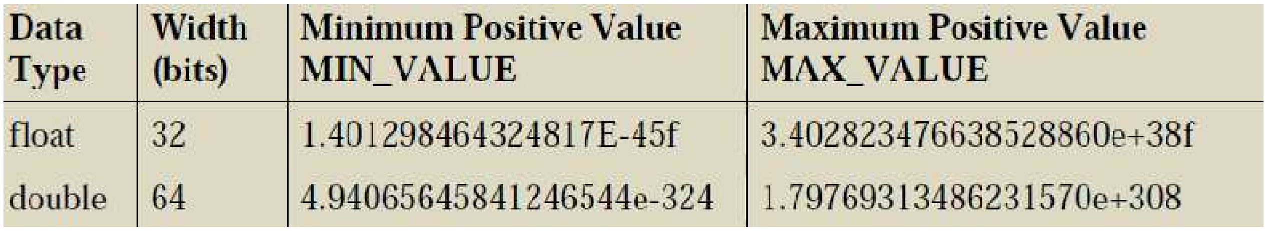
Range of character values



Data Type	Width (bits)	Minimum Unicode value	Maximum Unicode value
char	16	0x0 (\u0000)	0xffff (\uffff)

Characters are represented by the data type char. Their values are unsigned integers 16that denote all the 65536 (2) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters. The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

Floating-point Types



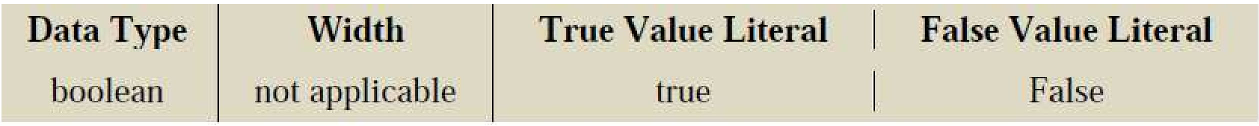
Data Type	Width (bits)	Minimum Positive Value MIN_VALUE	Maximum Positive Value MAX_VALUE
float	32	1.401298464324817E-45f	3.40282347663852886e+38f
double	64	4.94065645841246544e-324	1.79769313486231570e+308

Floating-point numbers are represented by the float and double data types.

Floating-point numbers conform to the IEEE 754-1985 binary floating-point standard. Above Table shows the range of values for positive floating-point numbers, but these apply equally to negative floating-point numbers with the '-' sign as prefix. Zero can be either 0.0 or -0.0.

Since the size for representation is finite, certain floating-point numbers can only be represented as approximations. For example, the value of the expression (1.0/3.0) is represented as an approximation due to the finite number of bits used.

Boolean Values



Data Type	Width	True Value Literal	False Value Literal
boolean	not applicable	true	False

The data type boolean represents the two logical values denoted by the literals true and false.

Boolean values are produced by all relational, conditional and boolean logical operators and are primarily used to govern the flow of control during program execution.

2.3 Variable Declarations

A variable stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can only store values of primitive data types and references to objects. Variables that store references to objects are called reference variables.

Declaring and Initializing Variables

Variable declarations are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. We show some examples of declaring variables that can store primitive values:

```
char a, b, c;        // a, b and c are character variables.
double area;          // area is a floating-point variable.
boolean flag;         // flag is a boolean variable.
```

The first declaration above is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also include initialization code to specify an appropriate initial alue for the variable:

```
int i = 10,           // i is an int variable with initial value 10.
  j = 101;            // j is an int variable with initial value 101.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

Object Reference Variables

An object reference is a value that denotes an object in Java. Such reference values can be stored in variables and used to manipulate the object denoted by the reference value.

A variable declaration that specifies a reference type (i.e., a class, an array, or an interface name) declares an object reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration

only specifies the name and the reference type. The declaration determines what objects a reference variable can denote. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

Pizza yummyPizza; // Variable yummyPizza can reference objects of class Pizza. Hamburger bigOne, // Variable bigOne can reference objects of class Hamburger, smallOne; // and so can variable smallOne.

It is important to note that the declarations above do not create any objects of class Pizza or Hamburger. The declarations only create variables that can store references to objects of these classes.

A declaration can also include an initializer to create an object whose reference can be assigned to the reference variable:

Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration with initializer.

The reference variable yummyPizza can reference objects of class Pizza. The keyword new, together with the constructor call Pizza("Hot&Spicy"), creates an object of class Pizza. The reference to this object is assigned to the variable yummyPizza. The newly created object of class Pizza can now be manipulated through the reference stored in this variable.

Lifetime of Variables

Lifetime of a variable, that is, the time a variable is accessible during execution, is determined by the context in which it is declared. We distinguish between lifetime of variables in three contexts:

• Instance variables– members of a class and created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the state of the object. Instance variables exist as long as the object they belong to exists.

• Static variables– also members of a class, but not created for any object of the class and, therefore, belong only to the class. They are created when the class is loaded at runtime, and exist as long as the class exists. · Local variables (also called method automatic variables)– declared in methods and in blocks and created for each execution of the method or block. After the execution of the method or block completes, local (non-final) variables are no longer accessible.

2.4 The main() Method

The Java interpreter executes a method called main in the class specified on the command line. Any class can have a main() method, but only the main() method of the class specified to the Java interpreter is executed to start a Java application. The main() method must have public accessibility so that the interpreter can call it. It is a static method belonging to the class, so that no object of the class is required to start the execution. It does not return a value, that is, it is declared void . It always has an array of String objects as its only formal parameter. This array contains any arguments passed to the program on the command line. All this adds up to the following definition of the main() method:

```
public static void main(String[] args) {

   // ...

}
```

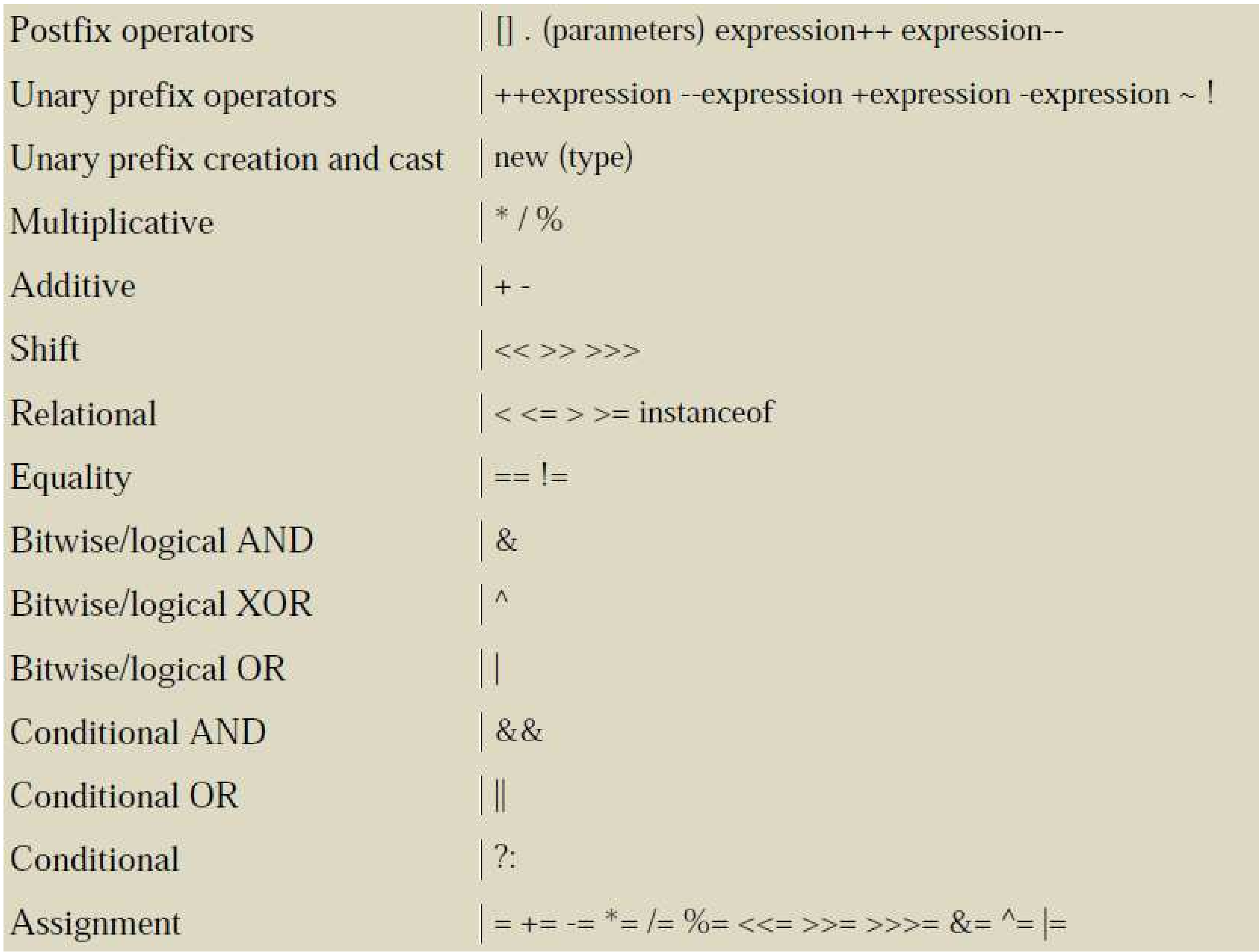
The above requirements do not exclude specification of additional modifiers or any throws clause. The main() method can also be overloaded like any other method. The Java interpreter ensures that the main() method, that complies with the above definition is the starting point of the program execution.

03

Operators and Assignments

3.1 Precedence and Associativity Rules for Operators

Precedence and associativity rules are necessary for deterministic evaluation of expressions. The operators are summarized in the following table 3.1.



Postfix operators	[] . (parameters) expression++ expression--
Unary prefix operators	++expression --expression +expression -expression ~ !
Unary prefix creation and cast	new (type)
Multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< <= > >= instanceof
Equality	== !=
Bitwise/logical AND	&
Bitwise/logical XOR	^
Bitwise/logical OR	|
Conditional AND	&&
Conditional OR	||
Conditional	?:
Assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=

Table 3.1: Operator Summary

The following remarks apply to Table 3.1:

- The operators are shown with decreasing precedence from the top of the table.
- Operators within the same row have the same precedence.
- Parentheses, ( ), can be used to override precedence and associativity.

· The unary operators, which require one operand, include the postfix increment (++) and decrement (--) operators from the first row, all the prefix operators (+, -, ++, --, ~, !) in the second row, and the prefix operators (object creation operator new, cast operator (type)) in the third row.
· The conditional operator (? :) is ternary, that is, requires three operands.
· All operators not listed above as unary or ternary, are binary, that is, require two operands.
• All binary operators, except for the relational and assignment operators, associate from left to right. The relational operators are nonassociative.
• Except for unary postfix increment and decrement operators, all unary operators, all assignment operators, and the ternary conditional operator associate from right to left.

Precedence rules are used to determine which operator should be applied first if there are two operators with different precedence, and these follow each other in the expression. In such a case, the operator with the highest precedence is applied first. 2 + 3 * 4 is evaluated as 2 + (3 * 4) (with the result 14) since * has higher precedence than +.

Associativity rules are used to determine which operator should be applied first if there are two operators with the same precedence, and these follow each other in the expression.

Left associativity implies grouping from left to right: 1 + 2 - 3 is interpreted as ((1 + 2) - 3), since the binary operators + and - both have same precedence and left associativity.

Right associativity implies grouping from right to left: - - 4 is interpreted as (- (- 4)) (with the result 4), since the unary operator - has right associativity.

The precedence and associativity rules together determine the evaluation order of the operators.

3.2 Evaluation Order of Operands

In order to understand the result returned by an operator, it is important to understand the evaluation order of its operands. Java states that the operands of operators are evaluated from left to right.

Java guarantees that all operands of an operator are fully evaluated before the operator is applied. The only exceptions are the short-circuit conditional operators &&, ||, and ?:.

In the case of a binary operator, if the left-hand operand causes an exception the right-hand operand is not evaluated. The evaluation of the left-hand operand can have side effects that can influence the value of the right-hand operand. For example, in the following code:

int b = 10;

System.out.println((b=3) + b);

the value printed will be 6 and not 13. The evaluation proceeds as follows: (b=3) + b

3 + b b is assigned the value 3

3 + 3

6

The evaluation order also respects any parentheses, and the precedence and associativity rules of operators.

3.3 Conversions

In this section we discuss the different kinds of type conversions and list the contexts in which these can occur. Some type conversions must be explicitly stated in the program, while others are done implicitly. Some type conversions can be checked at compile time to guarantee their validity at runtime, while others will require an extra check at runtime.

Unary Cast Operator: (type)

Java, being a strongly typed language, checks for type compatibility (i.e., checks if a type can substitute for another type in a given context) at compile time. However, some checks are only possible at runtime (for example, which type of object a reference actually denotes during execution). In cases where an operator would have incompatible operands (for example, assigning a double to an int), Java demands that a cast be used to explicitly indicate the type conversion. The cast construct has the following syntax:

(<type>) <expression>

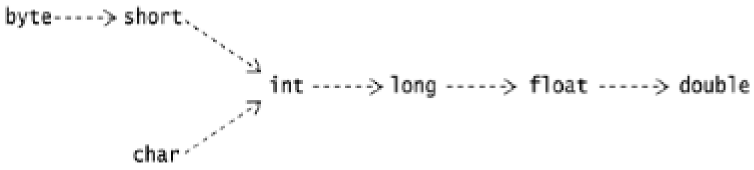
The cast (<type>) is applied to the value of the <expression>. At runtime, a cast results in a new value of <type>, which best represents the value of the <expression> in the old type. We use the term casting to mean applying the cast operator for explicit type conversion.

Casting can be applied to primitive values as well as references. Casting between primitive data types and reference types is not permitted. Boolean values cannot be cast to other data values, and vice versa. The reference literal null can be cast to any reference type.

Narrowing and Widening Conversions

For the primitive data types, the value of a narrower data type can be converted to a value of a broader data type without loss of information. This is called a widening primitive conversion. Widening conversions to the next broader type for primitive data types are summarized in Figure 3.1. The conversions shown are transitive. For example, an int can be directly converted to a double without first having to convert it to a long and a float.

Figure 3.1. Widening Numeric Conversions



Converting from a broader data type to a narrower data type is called a narrowing primitive conversion, which can result in loss of magnitude information. Any conversion which is not a widening conversion according to Figure 3.1 is a narrowing conversion. Note that all conversions between char and the two integer types byte and short are considered narrowing conversions: the reason being that the conversions between the unsigned type char and the signed types byte or short can result in loss of information.

Widening and narrowing conversions are also defined for reference types. Conversions up the inheritance hierarchy are called widening reference conversions (a.k.a. upcasting). Conversions down the inheritance hierarchy are called narrowing reference conversions (a.k.a. downcasting). Both narrowing and widening conversions can be either explicit (requiring a cast) or implicit. Widening conversions are usually done implicitly, whereas narrowing conversions typically require a cast. It is not illegal to use a cast for a widening conversion. However, the compiler will flag any conversions that require a cast.

Numeric Promotions

Numeric operators only allow operands of certain types. Numeric promotion is implicitly applied on the operands to convert them to permissible types. Distinction is made between unary and binary numeric promotion.

Unary Numeric Promotion

Unary numeric promotion states that

If the single operand of the operator has a type narrower than int, it is converted to int by an implicit widening primitive conversion; otherwise, it is not converted.

In other words, unary numeric promotion converts operands of byte, short, and char to int by applying an implicit widening conversion, but operands of other numeric types are not affected.

Unary numeric promotion is applied in the following contexts:

- operand of the unary arithmetic operators + and –
- operand of the unary integer bitwise complement operator ~
- during array creation; for example, new int[20], where the dimension expression (in this case 20) must evaluate to an int value

- indexing array elements; for example, table['a'], where the index expression (in this case 'a') must evaluate to an int value
- individual operands of the shift operators <<, >> and >>>

Binary Numeric Promotion

Binary numeric promotion implicitly applies appropriate widening primitive conversions so that a pair of operands have the broadest numeric type of the two, which is always at least int. Given T to be the broadest numeric type of the two operands, the operands are promoted as follows under binary numeric promotion. If T is broader than int, both operands are converted to T; otherwise, both operands are converted to int.

This means that byte, short, and char are always converted to int at least.

Binary numeric promotion is applied in the following contexts:

- operands of the arithmetic operators *, /, %, +, and –
- operands of the relational operators <, <=, >, and >=
- operands of the numerical equality operators == and !=
- operands of the integer bitwise operators &, ^, and |

Type Conversion Contexts

Type conversions can occur in the following contexts:

- assignments involving primitive data types and reference types
- method invocation involving parameters of primitive data types and reference types
- arithmetic expression evaluation involving numeric types
- string concatenation involving objects of class String and other data types

3.4 Simple Assignment Operator =

The assignment statement has the following syntax:
<variable> = <expression>
which can be read as "the destination, <variable>, gets the value of the source, <expression>". The previous value of the destination variable is overwritten by the assignment operator =.

The destination <variable> and the source <expression> must be type compatible. The destination variable must also have been declared. Since variables can store either primitive data values or object references, <expression> evaluates to either a primitive data value or an object reference.

Assigning Primitive Values

The following examples illustrate assignment of primitive values:

```
 int j, k;
j = 10;          // j gets the value 10.
j = 5;          // j gets the value 5. Previous value is overwritten.
 k = j;          // k gets the value 5.
```

The assignment operator has the lowest precedence, allowing the expression on the right-hand side to be evaluated before assignment.

```
int i;
i = 5;          // i gets the value 5.

i = i + 1;       // i gets the value 6. + has higher precedence than =. i = 20 - i * 2;   // i gets the value
8: (20 - (i * 2))
```

Assigning References

Copying references by assignment creates aliases. The following example recapitulates that discussion:

```
Pizza pizza1 = new Pizza("Hot&Spicy");

Pizza pizza2 = new Pizza("Sweet&Sour");

 pizza2 = pizza1;
```

Multiple Assignments

The assignment statement is an expression statement, which means that application of the binary assignment operator returns the value of the expression on the right-hand side.

```
int j, k;

j = 10;          // j gets the value 10 which is returned

 k = j;          // k gets the value of j, which is 10, and this value is returned
```

The last two assignments can be written as multiple assignments, illustrating the right associativity of the assignment operator.

```
k = j = 10;      // (k = (j = 10))
```

Multiple assignments are equally valid with references.

```
Pizza pizzaOne, pizzaTwo;
```

```
pizzaOne = pizzaTwo = new Pizza("Supreme"); // Aliases.
```

The following example shows the effect of operand evaluation order:

```
int[] a = {10, 20, 30, 40, 50}; // an array of int int index = 4;

a[index] = index = 2;
```

What is the value of index, and which array element a[index] is assigned a value in the multiple assignment statement? The evaluation proceeds as follows:

```
 a[index] = index = 2;
a[4]     = index = 2;
a[4]     = (index = 2);      // index gets the value 2. = is right associative.
a[4]     =      2;           // The value of a[4] is changed from 50 to 2.
```

Numeric Type Conversions on Assignment

If the destination and source have the same type in an assignment, then, obviously, the source and the destination are type compatible, and the source value need not be converted. Otherwise, if a widening primitive conversion is permissible, then the widening conversion is applied implicitly, that is, the source type is promoted to the destination type in an assignment context.

```
// Implicit Widening Primitive Conversions
 int    smallOne = 1234;
long   bigOne   = 2000;            // Implicit widening: int to long.
double largeOne = bigOne;          // Implicit widening: long to double.
double hugeOne  = (double) bigOne;  // Cast redundant but allowed.
```

Integer values widened to floating-point values can result in loss of precision. Precision relates to the number of significant bits in the value, and must not be confused with magnitude, which relates how big a value can be represented. In the next example, the precision of the least significant bits of the long value may be lost when converting to a float value.

```
long bigInteger = 98765432112345678L;
float realNo = bigInteger;  // Widening but loss of precision: 9.8765436E16
```

Additionally, implicit narrowing primitive conversions on assignment can occur in cases where all of the following conditions are fulfilled:

· the source is a constant expression of either byte, short, char, or int type

· the destination type is either byte, short, or char type

· the value of the source is determined to be in the range of the destination type at compile time

```
// Above conditions fulfilled for implicit narrowing primitive conversions.

short s1 = 10;          // int value in range.
short s2 = 'a';         // char value in range.
char c1 = 32;           // int value in range.
char c2 = (byte)35;     // byte value in range. int value in range, without cast. byte b1 = 40;           //
int value in range.
byte b2 = (short)40;    // short value in range. int value in range, without cast. final int i1 = 20;
byte b3 = i1;           // final value of i1 in range.

// Above conditions not fulfilled for implicit narrowing primitive conversions. // Explicit cast
required.
int i2 = -20;
final int i3 = i2;
final int i4 = 200;
short s3 = (short) i2;   // Not constant expression.
char c3 = (char) i3;     // final value of i3 not determinable.
char c4 = (char) i2;     // Not constant expression.
byte b4 = (byte) 128;    // int value not in range.
byte b5 = (byte) i4;     // final value of i4 not in range.
```

All other narrowing primitive conversions will produce a compile-time error on assignment, and will explicitly require a cast.

Floating-point values are truncated when converted to integral values.

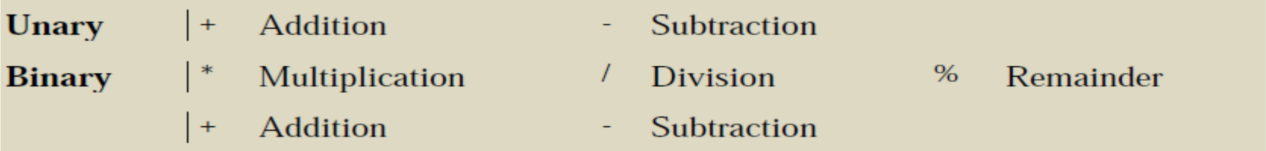
```
// Explicit narrowing primitive conversions requiring cast.
// The value is truncated to fit the size of the destination type.
float huge   = (float) 1.7976931348623157d; // double to float.
long  giant  = (long) 4415961481999.03D;   // (1) double to long.
int   big    = (int) giant;                // (2) long to int.
short small  = (short) big;                // (3) int to short.
byte  minute = (byte) small;               // (4) short to byte.
char  symbol = (char) 112.5F;              // (5) float to char.
```

3.5 Arithmetic Operators: *, /, %, +, -

The arithmetic operators are used to construct mathematical expressions as in algebra. Their operands are of numeric type (which includes the char type).

Arithmetic Operator Precedence and Associativity

In the following table, the precedence of the operators is in decreasing order, starting from the top row, which has the highest precedence. Unary subtraction has higher precedence than multiplication. The operators in the same row have the same precedence. Binary multiplication, division, and remainder operators have the same precedence. The unary operators have right associativity, and the binary operators have left associativity.



Unary	+	Addition	-	Subtraction		
Binary	*	Multiplication	/	Division	%	Remainder
	+	Addition	-	Subtraction		

Evaluation Order in Arithmetic Expressions

Java guarantees that the operands are fully evaluated from left to right before an arithmetic binary operator is applied. Of course, if evaluation of an operand causes an exception, the subsequent operands will not be evaluated.

In the expression a + b * c, the operand a will always be fully evaluated before the operand b, which will always be fully evaluated before the operand c. However, the multiplication operator * will be applied before the addition operator +, respecting the precedence rules. Note that a, b, and c can be arbitrary arithmetic expressions that have been determined to be the operands of the operators.

Range of Numeric Values

This range is given by the constants named MAX_VALUE and MIN_VALUE, which are defined in each numeric wrapper class.

The arithmetic operators are overloaded, meaning that the operation of an operator varies depending on the type of its operands. Floating-point arithmetic is performed if any operand of an operator is of floating-point type, otherwise, integer arithmetic is performed.

Values that are out-of-range or are results of invalid expressions, are handled differently depending on whether integer or floating-point arithmetic is performed.

Integer Arithmetic

Integer arithmetic always returns a value that is in range, except in the case of integer division by zero and remainder by zero, which causes an ArithmeticException (see the division operator / and the remainder operator % below). A valid value does not necessarily mean that the result is correct, as demonstrated by the following examples:

```
int tooBig   = Integer.MAX_VALUE + 1;    // -2147483648 which is
 Integer.MIN_VALUE.
int tooSmall = Integer.MIN_VALUE - 1;    //  2147483647 which is
Integer.MAX_VALUE.
```

The results above should be values that are out-of-range. However, integer arithmetic wraps if the result is out-of-range, that is, the result is reduced modulo in the range. In order to avoid wrapping of out-of-range values, programs should either use explicit checks or a wider type. If the type long is used in the examples above, the results would be correct in the long range:

```
long notTooBig   = Integer.MAX_VALUE + 1L;   //  2147483648L in range.
 long notTooSmall = Integer.MIN_VALUE - 1L;   // -2147483649L in range.
```

Floating-point Arithmetic

Certain floating-point operations result in values that are out-of-range. Typically, adding or multiplying two very large floating-point numbers can result in an out-of-range value which is represented by Infinity. Attempting floating-point division by

zero also returns infinity. The examples below show how this value is printed as signed infinity.

```
System.out.println( 4.0 / 0.0);        // Prints:  Infinity
System.out.println(-4.0 / 0.0);        // Prints: -Infinity
```

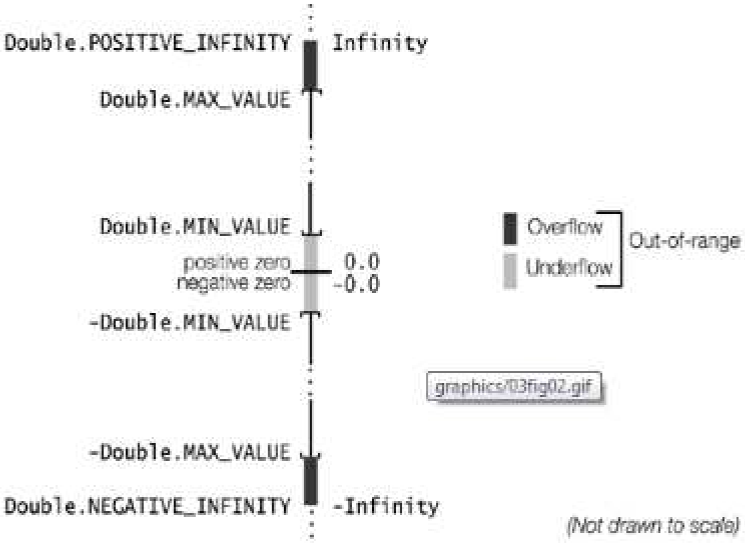


Fig: Overflow and Underflow in Floating-point Arithmetic

Both positive and negative infinity represent overflow to infinity, that is, the value is too large to be represented as a double or float. Signed infinity is represented by named constants POSITIVE_INFINITY and NEGATIVE_INFINITY, in the wrapper classes java.lang.Float and java.lang.Double. A value can be compared with these constants to detect overflow.

Floating-point arithmetic can also result in underflow to zero, that is, the value is too small to be represented as a double or float. Underflow occurs in the following situations:

- the result is between Double.MIN_VALUE (or Float.MIN_VALUE) and zero; for example, the result of (5.1E-324 - 4.9E-324). Underflow then returns positive zero 0.0 (or 0.0F).

- the result is between -Double.MIN_VALUE (or -Float.MIN_VALUE) and zero; for example, the result of (-Double.MIN_VALUE * 1E-1). Underflow then returns negative zero -0.0 (or -0.0F).

Negative zero compares equal to positive zero, i.e. (-0.0 == 0.0) is true.

Certain operations have no mathematical result, and are represented by NaN (Not a Number), for example calculating the square root of -1. Another example is (floating-point) dividing zero by zero:

```
System.out.println(0.0 / 0.0);      // Prints:
```

NaN is represented by the constant named NaN in the wrapper classes java.lang.Float and java.lang.Double. Any operation involving NaN produces NaN. Any comparison (except inequality !=) involving NaN and any other value (including NaN) returns false. An inequality comparison of NaN with another value (including NaN) always returns true. However, the recommended way of checking a value for NaN is to use the static method isNaN() defined in both wrapper classes, java.lang.Float and java.lang.Double.

Strict Floating-Point Arithmetic: strictfp

Although floating-point arithmetic in Java is defined in accordance with the IEEE-754 32-bit (float) and 64-bit (double) standard formats, the language does allow JVM implementations to use other extended formats for intermediate results. This means that floating-point arithmetic can give different results on such JVMs, with possible loss of precision. Such a behavior is termed non-strict, in contrast to being strict and adhering to the standard formats.

To ensure identical results are produced on all JVMs, the keyword strictfp can be used to enforce strict behavior for floating-point arithmetic. The modifier strictfp can be applied to classes, interfaces, and methods. A strictfp method ensures that all code in the method is executed strictly. If a class or interface is declared to be strictfp, then all code (in methods, initializers, and nested classes and interfaces) is executed strictly. If the expression is determined to be in a strictfp construct, it is executed strictly. However, note that strictness is not inherited by the subclasses or subinterfaces. Constant expressions are always evaluated strictly at compile time.

Unary Arithmetic Operators: -, +

The unary operators have the highest precedence of all the arithmetic operators. The unary operator - negates the numeric value of its operand. The following example illustrates the right associativity of the unary operators: int value = - -10;   // (-(-10)) is 10

Notice the blank needed to separate the unary operators; otherwise, these would be interpreted as the decrement operator -- . The unary operator + has no effect on the evaluation of the operand value.

Multiplicative Binary Operators: *, /, %

Multiplication Operator: *

Multiplication operator * multiplies two numbers.

```
int    sameSigns     = -4    * -8;       // result:  32
 double oppositeSigns =  4.0  * -8.0;     // result: -32.0
int    zero          =  0    * -0;       // result:   0
```

Division Operator: /

The division operator / is overloaded. If its operands are integral, the operation results in integer division.

```
int    i1 = 4  / 5;   // result: 0
int    i2 = 8  / 8;   // result: 1
double d1 = 12 / 8;   // result: 1 by integer division. d1 gets the value 1.0.
```

Integer division always returns the quotient as an integer value, i.e. the result is truncated toward zero. Note that the division performed is integer division if the operands have integral values, even if the result will be stored in a floating-point type. An ArithmeticException is thrown when attempting integer division with zero, meaning that integer division by zero is an illegal operation. If any of the operands is a floating-point type, the operation performs floating-point division.

```
double d2 = 4.0 / 8;      // result: 0.5
double d3 = 8 / 8.0;      // result: 1.0
double d4 = 12.0F / 8;    // result: 1.5F

double result1 = 12.0 / 4.0 * 3.0;      // ((12.0 / 4.0) * 3.0) which is 9
double result2 = 12.0 * 3.0 / 4.0;      // ((12.0 * 3.0) / 4.0) which is 9
```

Remainder Operator: %

In mathematics, when we divide a number (the dividend) by a another number (the divisor), the result can be expressed in terms of a quotient and a remainder. For example, dividing 7 by 5, the quotient is 1 and the remainder is 2. The remainder operator % returns the remainder of the division performed on the operands. int quotient = 7 / 5;   // Integer division operation: 1 int remainder = 7 % 5;   // Integer remainder operation: 2 For integer remainder operation, where only integer operands are involved, evaluation of the expression (x % y) always satisfies the following relation: x == (x / y) * y + (x % y)

In other words, the right-hand side yields a value that is always equal to the value of the dividend. The following examples show how we can calculate the remainder so that the above relation is satisfied:

Calculating (7 % 5):

7 == (7 / 5) * 5 + (7 % 5)

  == (1) * 5 + (7 % 5)

  == 5 + (7 % 5)

2 == (7 % 5)   i.e., (7 % 5) is equal to 2

Calculating (7 % -5):

7 == (7 / -5) * -5 + (7 % -5)

  == (-1) * -5 + (7 % -5)

  == 5 + (7 % -5)

2 == (7 % -5)   i.e., (7 % -5) is equal to 2

Calculating (-7 % 5):

-7 == (-7 / 5) * 5 + (-7 % 5)

   == (-1) * 5 + (-7 % 5)

   ==   -5 + (-7 % 5)

-2 ==   (-7 % 5)   i.e., (-7 % 5) is equal to -2

Calculating (-7 % -5):

-7 == (-7 / -5) * -5 + (-7 % -5)

```
  == (   1   ) * -5 + (-7 % -5)

  ==            -5 + (-7 % -5)

-2 ==                  (-7 % -5)  i.e., (-7 % -5) is equal to -2
```

The above relation shows that the remainder can only be negative if the dividend is negative, the sign of the divisor is irrelevant. A short-cut to evaluating the remainder involving negative operands is the following: ignore the signs of the operands, calculate the remainder, and negate the remainder if the dividend is negative.

```
 int  r0 =  7  %  7;     //  0
int  r1 =  7  %  5;     //  2
long r2 =  7L % -5L;    //  2L
int  r3 = -7  %  5;     // -2
long r4 = -7L % -5L;    // -2L
boolean relation = -7L == (-7L / -5L) * -5L + r4;  // true
```

An ArithmeticException is thrown when attempting integer remainder operation with zero.

Note that the remainder operator not only accepts integral operands, but floating-point operands as well. The floating-point remainder r is defined by the relation:

r == a - (b * q)

where a and b are the dividend and the divisor, respectively, and q is the integer quotient of (a/b). The following examples illustrate a floating-point remainder operation:

```
double  dr0 =  7.0  %  7.0;    //  0.0
float   fr1 =  7.0F %  5.0F;   //  2.0F
double  dr1 =  7.0  % -5.0;    //  2.0
float   fr2 = -7.0F %  5.0F;   // -2.0F
double  dr2 = -7.0  % -5.0;    // -2.0
```

boolean fpRelation = dr2 == (-7.0) - (-5.0) * (long)(-7.0 / -5.0); // true float   fr3 = -7.0F % 0.0F; // NaN

3.6 The Binary String Concatenation Operator +

The binary operator + is overloaded in the sense that the operation performed is determined by the type of the operands. When one of the operands is a String object, the other operand is implicitly converted to its string representation and string concatenation is performed. Non-String operands are converted as follows:

· For an operand of a primitive data type, its value is converted to a String object with the string representation of the value.

• Values like true, false, and null are represented by string representations of these literals. A reference variable with the value null also has the string representation "null" in this context.

• For all reference value operands, a string representation is constructed by calling the toString() method on the referred object. Most classes override this method from the Object class in order to provide a more meaningful string representation of their objects. The result of the concatenation is always a new String object.

```
String theName = " Uranium";
theName = " Pure" + theName;                // " Pure Uranium"
String trademark1 = 100 + "%" + theName;    // "100% Pure Uranium"
```

3.7 Variable Increment and Decrement Operators: ++, --

Variable increment (++) and decrement (--) operators come in two flavors: prefix and postfix. These unary operators have the side effect of changing the value of the arithmetic operand, which must evaluate to a variable. Depending on the operator used, the variable is either incremented or decremented by 1. These operators are very useful for updating variables in loops where only the side effect of the operator is of interest.

Increment Operator ++

Prefix increment operator has the following semantics: ++i adds 1 to i first, then uses the new value of i as the value of the expression. It is equivalent to the following statements.

```
i += 1;
result = i;
return result;
```

Postfix increment operator has the following semantics:

j++ uses the current value of j as the value of the expression first, then adds 1 to j. It is equivalent to the following statements:

```
result = j;
j += 1;
return result;
```

Decrement Operator --

Prefix decrement operator has the following semantics: --i subtracts 1 from i first, then uses the new value of i as the value of the expression. Postfix decrement operator has the following semantics: j-- uses the current value of j as the value of the expression first, then subtracts 1 from j.

Examples of Increment and Decrement Operators

```
 // (1) Prefix order: increment operand before use.
 int i = 10;
int k = ++i + --i;  // ((++i) + (--i)). k gets the value 21 and i becomes 10.
--i;                // Only side effect utilized. i is 9. (expression statement)
// (2) Postfix order: increment operand after use. long i = 10;
long k = i++ + i--;  // ((i++) + (i--)). k gets the value 21L and i becomes 10L. i++;                  //
Only side effect utilized. i is 11L. (expression statement)
```

An increment or decrement operator, together with its operand can be used as an expression statement.

Execution of the assignment in the second declaration in (1) proceeds as follows :

```
k = ((++i) + (--i))         Operands are determined.
k = ( 11   + (--i))         i now has the value 11.
 k = ( 11   +  10)          i now has the value 10.
k = 21
```

Expressions where variables are modified multiple times during the evaluation should be avoided, because the order of evaluation is not always immediately apparent.

We cannot associate increment and decrement operators. Given that a is a variable, we cannot write (++(++a)). The reason is that any operand to ++ must evaluate to a variable, but the evaluation of (++a) results in a value.

In the case where the operand is of type char, byte, or short, both binary numeric promotion and an implicit narrowing conversion are performed to achieve the side effect of modifying the value of the operand. In the example below, the int value of (++b) (i.e., 11), is assigned to int variable i. The side effect of incrementing the value of byte variable b requires binary numeric promotion to perform int addition, followed by an implicit narrowing conversion of the int value to byte.

```
 byte b = 10;
int  i = ++b;        // i is 11, and so is b.
```

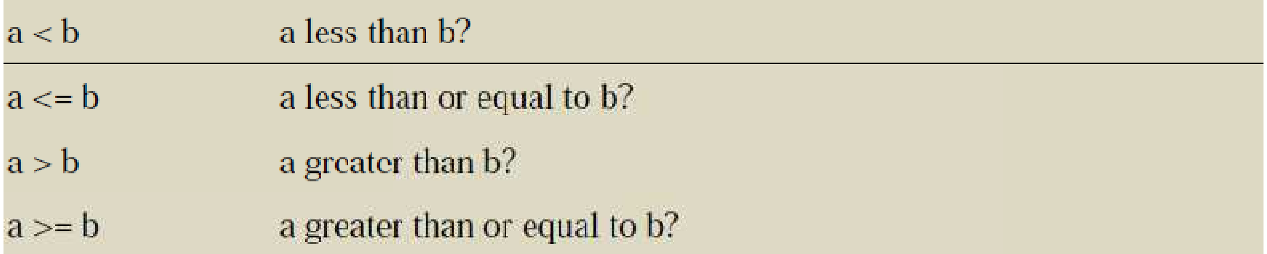
The increment and decrement operators can also be applied to floating-point operands. In the example below, the side effect of the ++ operator is overwritten by the assignment.

```
double x = 4.5;
```

```
x = x + ++x;        // x gets the value 10.0.
```

3.8 Relational Operators: <, <=, >, >=

Given that a and b represent numeric expressions, the relational (also called comparison) operators are defined as shown in the following table.



a < b	a less than b?
a <= b	a less than or equal to b?
a > b	a greater than b?
a >= b	a greater than or equal to b?

All relational operators are binary operators, and their operands are numeric expressions. Binary numeric promotion is applied to the operands of these operators. The evaluation results in a boolean value. Relational operators have precedence lower than arithmetic operators, but higher than that of the assignment operators.

```
double  hours = 45.5;
boolean overtime = hours >= 35.0;    // true.
 boolean order = 'A' < 'a'; // true. Binary numeric promotion applied.
```

Relational operators are nonassociative. Mathematical expressions like must be written using relational and boolean logical/conditional operators.

$a \leq b \leq$

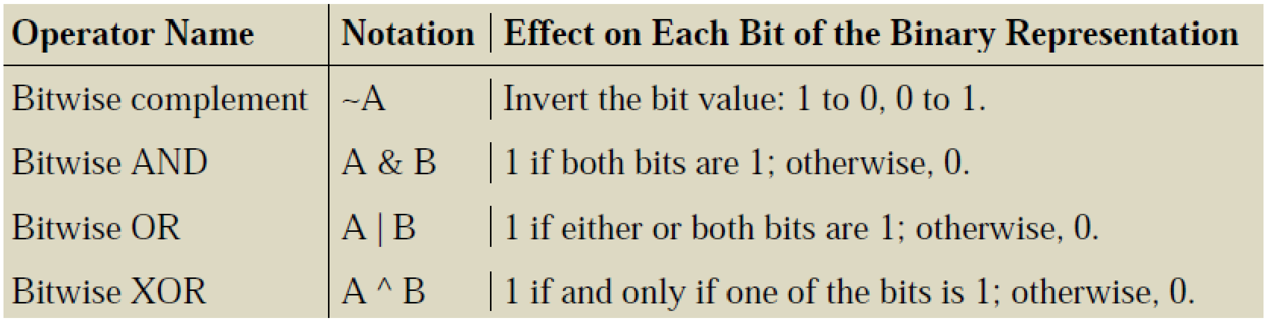
```
 int a = 1, b = 7, c = 10;
boolean valid1 = a <= b <= c;            // (1) Illegal.
boolean valid2 = a <= b && b <= c;       // (2) OK.
```

Since relational operators have left associativity, the evaluation of the expression a <= b <= c at (1) in the examples above would proceed as follows: ((a <= b) <= c). Evaluation of (a <= b) would yield a boolean value that is not permitted as an operand of a relational operator, that is, (<boolean value> <= c) would be illegal.

3.9 Integer Bitwise Operators: ~, &, |, ^

Integer bitwise operators include the unary operator ~ (bitwise complement) and the binary operators & (bitwise AND), | (bitwise inclusive OR), and ^ (bitwise exclusive OR, a.k.a. bitwise XOR).

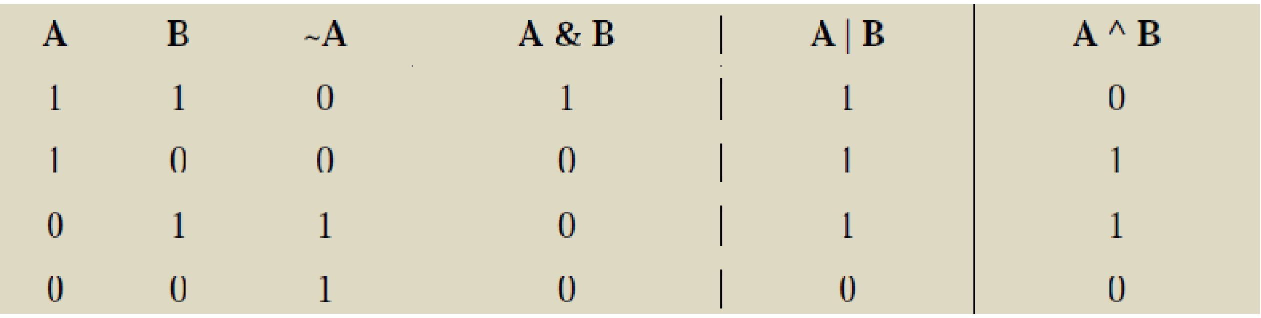
The binary bitwise operators perform bitwise operations between corresponding individual bit values in the operands. Unary numeric promotion is applied to the operand of the unary bitwise complement operator ~, and binary numeric promotion is applied to the operands of the binary bitwise operators. The result is a new integer value of the promoted type, which can only be either int or long. Given that A and B are corresponding bit values (either 0 or 1) in the left-hand and right-hand operands, respectively, these bitwise operators are defined as shown in the following table. The operators are listed in decreasing precedence order.



Operator Name	Notation	Effect on Each Bit of the Binary Representation
Bitwise complement	~A	Invert the bit value: 1 to 0, 0 to 1.
Bitwise AND	A & B	1 if both bits are 1; otherwise, 0.
Bitwise OR	A | B	1 if either or both bits are 1; otherwise, 0.
Bitwise XOR	A ^ B	1 if and only if one of the bits is 1; otherwise, 0.

The operators &, |, and ^ can also be applied to boolean operands to perform boolean logical operations.

Result of Applying Bitwise Operators :



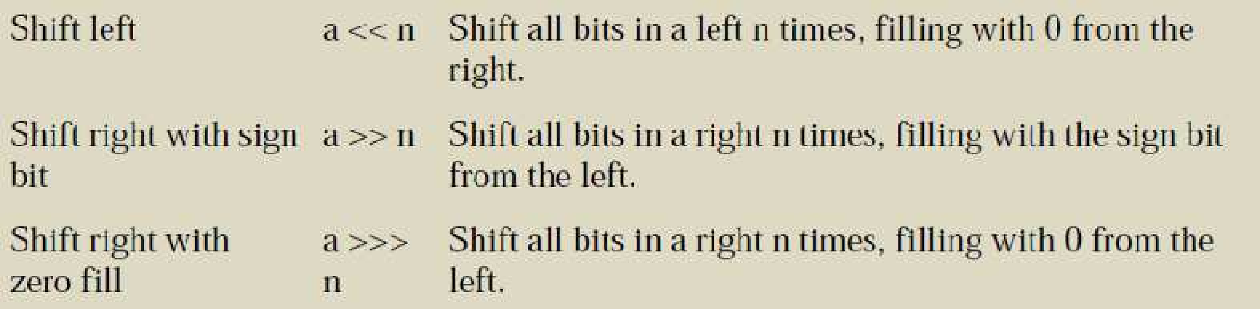
A	B	~A	A & B	A | B	A ^ B
1	1	0	1	1	0
1	0	0	0	1	1
0	1	1	0	1	1
0	0	1	0	0	0

3.10 Shift Operators: <<, >>, >>>

The binary shift operators form a new value by shifting bits either left or right a specified number of times in a given integral value. The number of shifts (also called the shift distance) is given by the right-hand operand, and the value that is to be shifted is given by the left-hand operand. Note that unary numeric promotion is applied to each operand individually. The value returned has the promoted type of the left-hand operand. Also, the value of the left-hand operand is not affected by applying the shift operator.

The shift distance is calculated by AND-ing the value of the right-hand operand with a mask value of 0x1f (31) if the left-hand has the promoted type int, or using a mask value of 0x3f (63) if the left-hand has the promoted type long. This effectively means masking the five lower bits of the right-hand operand in the case of an int left-hand operand, and masking the six lower bits of the right-hand operand in the case of a long left-hand operand. Thus, the shift distance is always in the range 0 to 31 when the promoted type of left-hand operand is int, and in the range 0 to 63 when the promoted type of left-hand operand is long.

Given that a contains the value whose bits are to be shifted and n specifies the number of bits to shift, the bitwise operators are defined in the following table. It is implied that the value n in the given table is subject to the shift distance calculation outlined above, and that the shift operations are always performed on the value of the left-hand operand represented in 2's complement.



Shift left	a << n	Shift all bits in a left n times, filling with 0 from the right.
Shift right with sign bit	a >> n	Shift all bits in a right n times, filling with the sign bit from the left.
Shift right with zero fill	a >>> n	Shift all bits in a right n times, filling with 0 from the left.

Since char, byte and short operands are promoted to int, the result of applying these bitwise operators is always either an int or a long value. Care must be taken in employing a cast to narrow the resulting value, as this can result in loss of information as the upper bits are chopped off during conversion. Note that regardless of the promotion of the values in the operands or determination of the shift distance, the operands a and n are not affected by these three shift operators. However, the shift compound assignment operators, discussed in this section, can change the value of the left-hand operand a. Bit values shifted out (falling off) from bit 0 or the most significant bit are lost. Since bits can be shifted both left and right, a positive value when shifted can result in a negative value and vice versa.

3.11 The Conditional Operator: ?

The ternary conditional operator allows conditional expressions to be defined. The operator has the following syntax:

<condition> ? <expression1> : <expression2>

If the boolean expression <condition> is true then <expression1> is evaluated; otherwise, <expression2> is evaluated. Of course, <expression1> and <expression2> must evaluate to values of compatible types. The value of the expression evaluated is returned by the conditional expression.

```
boolean leapYear = false;
int daysInFebruary = leapYear ? 29 : 28;   // 28
```

The conditional operator is the expression equivalent of the if-else statement. The conditional expression can be nested and the conditional operator associates from right to left:

(a?b?c?d:e:f:g) evaluates as (a?(b?(c?d:e):f):g)

3.12 Other Operators: new, [], instanceof

The new operator is used to create objects, that is, instances of classes and arrays. It is used with a constructor call to instantiate classes, and with the [] notation to create arrays. It is also used to instantiate anonymous arrays and anonymous classes.

```
Pizza onePizza = new Pizza();       // Create an instance of Pizza class.
```

The [] notation is used to declare and construct arrays and also to access array elements.

```
int[] anArray = new int[5];// Declare and construct an int array of 5 elements.
anArray[4] = anArray[3];   // Element at index 4 gets value of element at index 3.
```

The boolean, binary, and infix operator instanceof is used to test an object's type

```
Pizza myPizza = new Pizza();
boolean test1 = myPizza instanceof Pizza;  // True.
 boolean test2 = "Pizza" instanceof Pizza;  // Compile error. String //is not Pizza.
boolean test3 = null instanceof Pizza;     // Always false. null not //an instance.
```

04

Declarations and Access Controls

4.1 Arrays

An array is a data structure that defines an indexed collection of a fixed number of

homogeneous data elements. This means that all elements in the array have the
same data type. A position in the array is indicated by a non-negative integer value called the index. An element at a given position in the array is accessed using the index. The size of an array is fixed and cannot increase to accommodate more elements.

In Java, arrays are objects. Arrays can be of primitive data types or reference types. In the former case, all elements in the array are of a specific primitive data type. In the latter case, all elements are references of a specific reference type. References in the array can then denote objects of this reference type or its subtypes. Each array object has a final field called length, which specifies the array size, that is, the number of elements the array can accommodate. The first element is always at index 0 and the last element at index n-1, where n is the value of the length field in the array.

Simple arrays are one-dimensional arrays, that is, a simple sequence of values. Since arrays can store object references, the objects referenced can also be array objects. This allows implementation of array of arrays.

Declaring Array Variables

An array variable declaration has either the following syntax: <element type>[] <array name>;

or

<element type> <array name>[];

where <element type> can be a primitive data type or a reference type. The array variable <array name> has the type <element type>[]. Note that the array size is not specified. This means that the array variable <array name> can be assigned an array of any length, as long as its elements have <element type>.

It is important to understand that the declaration does not actually create an array. It only declares a reference that can denote an array object.

```
 int anIntArray[], oneInteger;
Pizza[] mediumPizzas, largePizzas;
```

These two declarations declare anIntArray and mediumPizzas to be reference variables that can denote arrays of int values and arrays of Pizza objects, respectively. The variable largePizzas can denote an array of pizzas, but the variable oneInteger cannot denote an array of int values—it is simply an int variable.

When the [] notation follows the type, all variables in the declaration are arrays. Otherwise the [] notation must follow each individual array name in the declaration.

An array variable that is declared as a member of a class, but is not initialized to denote an array, will be initialized to the default reference value null. This default initialization does not apply to local reference variables and, therefore, does not apply to local array variables either. This should not be confused with initialization of the elements of an array during array construction.

Constructing an Array

An array can be constructed for a specific number of elements of the element type, using the new operator. The resulting array reference can be assigned to an array variable of the corresponding type.

<array name> = new <element type> [<array size>];

The minimum value of <array size> is 0 (i.e., arrays with zero elements can be constructed in Java). If the array size is negative, a NegativeArraySizeException is thrown.

Given the following array declarations:

```
int anIntArray[], oneInteger;

Pizza[] mediumPizzas, largePizzas;
```

the arrays can be constructed as follows:

```
anIntArray = new int[10];          // array for 10 integers
mediumPizzas = new Pizza[5];         // array of 5 pizzas
 largePizzas = new Pizza[3];         // array of 3 pizzas
```

The array declaration and construction can be combined.

<element type1>[] <array name> = new <element type2>[<array size>];

However, here array type <element type2>[] must be assignable to array type <element type1>[]. When the array is constructed, all its elements are initialized to the default value for <element type2>. This is true for both member and local arrays when they are constructed.

In all the examples below, the code constructs the array and the array elements are implicitly initialized to their default value. For example, the element at index 2 in array anIntArray gets the value 0, and the element at index 3 in array mediumPizzas gets the value null when the arrays are constructed.

```
int[] anIntArray = new int[10];          // Default element value: 0.
Pizza[] mediumPizzas = new Pizza[5];  // Default element value: null.
// Pizza class extends Object class
Object objArray = new Pizza[3];       // Default element value: null.
```

// Pizza class implements Eatable interface Eatable[] eatables = new Pizza[2];   // Default element value: null.

The value of the field length in each array is set to the number of elements specified during the construction of the array; for example, medium Pizzas.length has the value 5.

Once an array has been constructed, its elements can also be explicitly initialized individually; for example, in a loop. Examples in the rest of this section make heavy use of a loop to traverse through the elements of an array for various purposes.

Initializing an Array

Java provides the means of declaring, constructing, and explicitly initializing an array in one declaration statement:

```
<element type>[] <array name> = { <array initialize list> };
```

This form of initialization applies to member as well as local arrays. The <array initialize list> is a comma-separated list of zero or more expressions. Such an array initialization block results in the construction and initialization of the array.

```
int[] anIntArray = {1, 3, 49, 2, 6, 7, 15, 2, 1, 5};
```

The array anIntArray is declared as an array of ints. It is constructed to hold 10 elements (equal to the length of the list of expressions in the block), where the first element is initialized to the value of the first expression (1), the second element to the value of the second expression (3), and so on.

```
// Pizza class extends Object class
Object[] objArray = { new Pizza(), new Pizza(), null };
```

The array objArray is declared as an array of the Object class, constructed to hold three elements. The initialization code sets the first two elements of the array to refer to two Pizza objects, while the last element is initialized to the null reference. Note that the number of objects created in the above declaration statement is actually three: the array object with three references and the two Pizza objects.

The expressions in the <array initialize list> are evaluated from left to right, and the array name obviously cannot occur in any of the expressions in the list. In the examples above, the <array initialize list> is terminated by the right curly bracket, }, of the block. The list can also be legally terminated by a comma. The following array has length two, not three:

```
Topping[] pizzaToppings = { new Topping("cheese"), new Topping("tomato"), };
```

The declaration statement at (1) in the following code defines an array of four String objects, while the declaration statement at (2) should make it clear that a String object is not the same as an array of char.

```
// Array with 4 String objects

String[] pets = {"crocodiles", "elephants", "crocophants", "elediles"}; // (1)

// Array of 3 characters

char[] charArray = {'a', 'h', 'a'};    // (2) Not the same as "aha".
```

Using an Array

The whole array is referenced by the array name, but individual array elements are accessed by specifying an index with the [] operator. The array element access expression has the following syntax:

```
<array name> [<index expression>]
```

Each individual element is treated as a simple variable of the element type. The index is specified by the <index expression>, which can be any expression that evaluates to an non-negative int value. Since the lower bound of an array is always 0, the upper bound is one less than the array size, that is, (<array name>.length-1). The ith element in the array has index (i-1). At runtime, the index value is automatically checked to ensure that it is within the array index bounds. If the index value is less than 0 or greater than or equal to <array name>.length in an array element access expression, an ArrayIndexOutOfBoundsException is thrown. A program can either check the index explicitly or catch the exception, but an illegal index is typically an indication of a program bug.

In the array element access expression, the <array name> can, in fact, be any expression that returns a reference to an array. For example, the following expression accesses the element with the index 1 in the character array returned by a call to the toCharArray() method of the String class: "AHA".toCharArray()[1]. The array operator [] is used to declare array types (Topping[]), specify array size (new Toppings[3]), and to access array elements (toppings[1]). This operator is not used when the array reference is manipulated, for example in an array reference assignment or when the array reference is passed as an actual parameter in a method call.

Example 4.1 shows traversal of arrays. The loop at (3) initializes the local array trialArray declared at (2) five times with pseudo-random numbers (from 0.0 to 100.0), by calling the method randomize() at (5). The minimum value in the array is found by calling the method findMinimum() at (6), and is stored in the array storeMinimum declared at (1). The loop at (4) prints the minimum values from the trials. The start value of the loop variable is initially set to 0. The loop condition tests whether the loop variable is less than the length of the array; this guarantees that the index will not go out of bounds.

Example 4.1 Using Arrays

```
class Trials {
    public static void main(String[] args) {
        // Declare and construct the local arrays
 double[] storeMinimum = new double[5];             // (1)
 double[] trialArray = new double[15];              // (2)
  for (int i = 0; i < storeMinimum.length; ++i) {    // (3)
 // Initialize the array
            randomize(trialArray);
            // Find and store the minimum value
storeMinimum[i] = findMinimum(trialArray);         }
        // Print the minimum values                        (4)
for (int i = 0; i < storeMinimum.length; ++i)
 System.out.println(storeMinimum[i]);     }


    public static void randomize(double[] valArray) {      // (5)
 for (int i = 0; i < valArray.length; ++i)            valArray[i] = Math.random() * 100.0;     }
```

```
    public static double findMinimum(double[] valArray) {  // (6)

  // Assume the array has at least one element.

double minValue = valArray[0];

        for (int i = 1; i < valArray.length; ++i)

  minValue = Math.min(minValue, valArray[i]);

 return minValue;

    }

}
```

Possible output from the program:

```
6.756931310985048
5.364063199341363
8.359410202984296
8.858272848258109
9.759950059619849
```

Anonymous Arrays

As shown earlier in this section, the following declaration statement

```
 <element type1>[] <array name> = new <element type2>[<array size>]; // (1)

 int[] intArray = new int[5];
```

can be used to construct arrays using an array creation expression. The size of the array is specified in the array creation expression, which creates the array and initializes the array elements to their default values. On the other hand, the following declaration statement

```
<element type>[] <array name> = { <array initialize list> }; // (2)

int[] intArray = {3, 5, 2, 8, 6};
```

both creates the array and initializes the array elements to specific values given in the array initializer block. However, the array initialization block is not an expression.

Java has another array creation expression, called anonymous array, which allows the concept of the array creation expression from (1) and the array initializer block from (2) to be combined, to create and initialize an array object:

```
 new <element type>[] { <array initialize list> }
new int[] {3, 5, 2, 8, 6}
```

The construct has enough information to create a nameless array of a specific type. Neither the name of the array nor the size of the array is specified. The construct returns an array reference that can be assigned and passed as parameter. In particular, the following two examples of declaration statements are equivalent.

```
int[] intArray = {3, 5, 2, 8, 6};           // (1)
int[] intArray = new int[] {3, 5, 2, 8, 6}; // (2)
```

In (1), an array initializer block is used to create and initialize the elements. In (2), an anonymous array expression is used. It is tempting to use the array initialization block as an expression; for example, in an assignment statement as a short cut for assigning values to array elements in one go. However, this is illegal−instead, an anonymous array expression should be used.

```
int[] daysInMonth;
daysInMonth = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // Not ok.
daysInMonth = new int[] {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}; // ok.
```

The concept of anonymous arrays is similar to that of anonymous classes: they both combine the definition and instantiation of classes into one process.

In Example 4.2, an anonymous array is constructed at (1) and passed as parameter to the static method findMinimum() defined at (2). Note that no array name or array size is specified for the anonymous array.

Example 4.2 Using Anonymous Arrays

```
class AnonArray {
   public static void main(String[] args) {
      System.out.println("Minimum value: " +
 findMinimum(new int[] {3, 5, 2, 8, 6}));          // (1)     }

   public static int findMinimum(int[] dataSeq) {                  // (2)

  // Assume the array has at least one element.

 int min = dataSeq[0];
```

```
       for (int index = 1; index < dataSeq.length; ++index)
  if (dataSeq[index] < min)
              min = dataSeq[index];
       return min;
   }
}
```

Output from the program:

```
Minimum value: 2
```

4.2 Defining Classes

A class declaration introduces a new reference type. It has the following general syntax:

```
<class modifiers> class <class name>
            <extends clause> <implements clause>  // Class header
{ // Class body
    <field declarations>
    <method declarations>
    <nested class declarations>
    <nested interface declarations>
    <constructor declarations>
    <initializer blocks>
}
```

In the class header, the name of the class is preceded by the keyword class. In addition, the class header can specify the following information:

- scope or accessibility modifier
- additional class modifiers
- any class it extends
- any interfaces it implements

The class body can contain member declarations which comprise

- field declarations
- method declarations
- nested class and interface declarations

Members declared static belong to the class and are called static members, and non-static members belong to the objects of the class and are called instance members. In addition, the following can be declared in a class body:

- constructor declarations

- static and instance initializer blocks

The member declarations, constructor declarations and initializer blocks can appear in any order in the class body.

In order to understand what code is legal to declare in a class, we distinguish between static context and non-static context. A static context is defined by static methods, static field initializers, and static initializer blocks. A non-static context is defined by instance methods, constructors, non-static field initializers, and instance initializer blocks. By static code we mean expressions and statements in a static context, and similarly by non-static code we mean expressions and statements in a non-static context. One crucial difference between the two contexts is that static code cannot refer to non-static members.

4.3 Defining Methods

The general syntax of a method declaration is

```
<method modifiers> <return type> <method name> (<formal parameter list>)
   <throws clause> // Method prototype { // Method body
    <local variable declarations>
    <nested local class declarations>
    <statements>
}
```

In addition to the name of the method, the method prototype can specify the following information:

- scope or accessibility modifier
- additional method modifiers
- the type of the return value, or void if the method does not return any value
- a formal parameter list
- checked exceptions thrown by the method in a throws clause

The formal parameter list is a comma-separated list of parameters for passing information to the method when the method is invoked by a method call. An empty parameter list must be specified by ( ). Each parameter is a simple variable declaration consisting of its type and name:

```
<parameter modifier> <type> <parameter name>
```

The parameter names are local to the method

The signature of a method comprises the method name and the formal parameter list only.

The method body is a block containing the local declarations and the statements of the method. Like member variables, member methods can be characterized as:

- instance methods, which are discussed later in this chapter
- static methods, which are discussed later in this chapter

Statements

Statements in Java can be grouped into various categories. Variable declarations with explicit initialization of the variables are called declaration statements. Other basic forms of statements are control flow statements and expression statements.

An expression statement is an expression terminated by a semicolon. The expression is evaluated for its side effect and its value discarded. Only certain types of expressions have meaning as statements. They include the following:

- assignments
- increment and decrement operators
- method calls
- object creation expression with the new operator

A solitary semicolon denotes the empty statement that does nothing.

A block, {}, is a compound statement which can be used to group zero or more local declarations and statements. Blocks can be nested, since a block is a statement that can contain other statements. A block can be used in any context where a simple statement is permitted. The compound statement which is embodied in a block, begins at the left brace, {, and ends with a matching right brace, }. Such a block must not be confused with an array initialization block in declaration statements.

Instance Methods and Object Reference this

Instance methods belong to every object of the class and can only be invoked on objects. All members defined in the class, both static and non-static, are accessible in the context of an instance method. The reason is that all instance methods are passed an implicit reference to the current object, that is, the object on which the method is being invoked. The current object can be referenced in the body of the instance method by the keyword this. In the body of the method, the this reference can be used like any other object reference to access members of the object. In fact, the keyword this can be used in any non-static context. The this reference can be used as a normal reference to reference the current object, but the reference cannot be modified.

The this reference to the current object is useful in situations where a local variable hides, or shadows, a field with the same name. In Example 4.4, the two parameters noOfWatts and indicator in the constructor of the Light class have the same names as the fields in the class. The example also declares a local variable location, which has the same name as one of the fields. The reference this can be used to distinguish the fields from the local variables. At (1), the this reference is used to identify the field noOfWatts, which is assigned the value of the parameter noOfWatts. Without the this reference at (2), the value of the parameter indicator is assigned back to this parameter and not to the field by the same name, resulting in a logical error. Similarly at (3), without the this reference it is the local variable location that is assigned the value of the parameter site, and not the field by the same name.

Example 4.4 Using this Reference

```
class Light {
    // Fields
    int     noOfWatts;      // wattage
    boolean indicator;      // on or off
    String  location;       // placement

    // Constructor
    public Light(int noOfWatts, boolean indicator, String site) {
  String location;

        this.noOfWatts = noOfWatts;   // (1) Assignment to field.
  indicator = indicator;        // (2) Assignment to parameter.
  location = site;              // (3) Assignment to local variable.
  this.superfluous();           // (4)
        superfluous();                // equivalent to call at (4)
   }

    public void superfluous() { System.out.println(this); }  // (5)
    public static void main(String[] args) {
        Light light = new Light(100, true, "loft");
  System.out.println("No. of watts: " + light.noOfWatts);
System.out.println("Indicator: "    + light.indicator);
 System.out.println("Location: "     + light.location);     }
}
```

Output from the program:

```
Light@df6ccd
Light@df6ccd
No. of watts: 100
Indicator: false
```

Location: null

If a member is not shadowed by a local declarations, then the simple name member is considered a short-hand notation for this.member. In particular, the this reference can be used explicitly to invoke other methods in the class. This is illustrated at (4) in Example 4.4, where the method superfluous() is called. If, for some reason, a method needs to pass the current object to another method, it can do so using the this reference. This is illustrated at (5) in Example 4.4, where the current object is passed to the println() method. Note that the this reference cannot be used in a static context, as static code is not executed in the context of any object.

Method Overloading

Each method has a signature, which comprises the name of the method and the types and order of the parameters in the formal parameter list. Several method implementations may have the same name, as long as the method signatures differ. This is called method overloading. Since overloaded methods have the same name, their parameter lists must be different.

Rather than inventing new method names, method overloading can be used when the same logical operation requires multiple implementations. The Java 2 SDK APIs make heavy use of method overloading. For example, the class java.lang.Math contains an overloaded method min(), which returns the minimum of two numeric values.

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)
```

In the following examples, five implementations of the method methodA are shown:

```
void methodA(int a, double b) { /* ... */ }      // (1)
int  methodA(int a)           { return a; }      // (2)
int  methodA()                { return 1; }      // (3)
long methodA(double a, int b) { return b; }      // (4)
long methodA(int x, double y) { return x; }      // (5) Not OK.
```

The corresponding signatures of the five methods are as follows:

methodA(int, double)	1'
methodA(int)	2': Number of parameters.
methodA()	3': Number of parameters.
methodA(double, int)	4': Order of parameters.

```
methodA(int, double)                    5': Same as 1'.
```

The first four implementations of the method named methodA are overloaded correctly, each time with a different parameter list and, therefore, different signatures. The declaration at (5) has the same signature methodA(int, double) as the declaration at (1) and is, therefore, not a valid overloading of this method.

```
 void bake(Cake k)  { /* ... */ }           // (1)
void bake(Pizza p) { /* ... */ }           // (2)

int     halfIt(int a) { return a/2; }         // (3)
double  halfIt(int a) { return a/2.0; }       // (4) Not OK. Same signature.
```

The method named bake is correctly overloaded at (1) and (2), with two different signatures. In the implementation, changing just the return type (as shown at (3) and (4)), is not enough to overload a method, and will be flagged as a compile-time error. The parameter list in the definitions must be different. Overloaded methods should be considered as individual methods that just happen to have the same name. Methods with the same name are allowed since methods are identified by their signature. At compile time, the right implementation is chosen based on the signature of the method call. Only methods declared in the same class and those that are inherited by the class can be overloaded. Method overloading should not be confused with method overriding

4.4 Constructors

The main purpose of constructors is to set the initial state of an object when the object is created by using the new operator.

A constructor has the following general syntax:

```
 <accessibility modifier> <class name> (<formal parameter list>)
<throws clause> // Constructor header
 { // Constructor body
   <local variable declarations>
   <nested local class declarations>
   <statements>  }
```

Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

- Modifiers other than an accessibility modifier are not permitted in the constructor header.

• Constructors cannot return a value and, hence, cannot specify a return type, not even void, in the constructor header, but they can contain the simple form of the return statement in the constructor body.

· Constructor name must be the same as the class name.

Class names and method names exist in different namespaces. Thus, there are no name conflicts in Example 4.5, where a method declared at (2) has the same name as the constructor declared at (1). However, using such naming schemes is strongly discouraged.

Example 4.5 Namespaces

```
public class Name {

   Name() {                    // (1)
      System.out.println("Constructor");
   }

   void Name() {               // (2)
      System.out.println("Method");
   }
   public static void main(String[] args) {
                  new Name().Name();     // (3) Constructor call followed by method call.

   }
}
```

```
Output from the program:
Constructor
Method
```

Default Constructor

A default constructor is a constructor without any parameters. In other words, it has the following signature:

<class name>()

If a class does not specify any constructors, then an implicit default constructor is supplied for the class. The implicit default constructor is equivalent to the following implementation:

<class name>() { super(); } // No parameters. Calls superclass constructor.

The only action taken by the implicit default constructor is to call the superclass constructor. This ensures that the inherited state of the object is initialized properly. In addition, all instance variables in the object are set to the default value of their type.

In the following code, the class Light does not specify any constructors.

```
class Light {
   // Fields
   int     noOfWatts;       // wattage
   boolean indicator;       // on or off
   String  location;       // placement

   // No constructors
   //...
}

class Greenhouse {

   // ...

   Light oneLight = new Light();     // (1) Call of implicit default constructor. }
```

In the previous code, the following implicit default constructor is employed when a Light object is created at (1):

```
Light() { super(); }
```

Creating an object using the new operator with the implicit default constructor, as at (1), will initialize the fields of the object to their default values (i.e., the fields noOfWatts, indicator, and location in a Light object will be initialized to 0, false, and null, respectively).

A class can choose to provide an implementation of the default constructor. In the following example, the class Light provides an explicit default constructor at (1). Note that it has the same name as the class, and that it does not take any parameters.

```
class Light {
   // ...
   // Explicit Default Constructor
   Light() {                    // (1)
      noOfWatts = 50;
      indicator = true;

      location  = "X";
   }
   //...
}
```

```
class Greenhouse {
   // ...
   Light extraLight = new Light();   // (2) Call of explicit default constructor. }
```

The explicit default constructor ensures that any object created with the expression new Light(), as at (2), will have its fields noOfWatts, indicator and location initialized to 50, true and "X", respectively.

If a class defines any explicit constructors, it can no longer rely on the implicit default constructor to set the state of the objects. If such a class requires a default constructor, its implementation must be provided. In the example below, class Light only provides a non-default constructor at (1). It is called at (2) when an object of class Light is created with the new operator. Any attempt to call the default constructor will be flagged as a compile time error as shown at (3).

```
class Light {
   // ...
   // Only non-default Constructor
   Light(int noOfWatts, boolean indicator, String location) {          // (1)
this.noOfWatts = noOfWatts;
      this.indicator = indicator;
      this.location  = location;
   }
   //...
}
class Greenhouse {
   // ...
   Light moreLight  = new Light(100, true, "Greenhouse");   // (2) OK.
//  Light firstLight = new Light();                    // (3) Compile time error.
}
```

Overloaded Constructors

Like methods, constructors can also be overloaded. Since the constructors in a class all have the same name as the class, their signatures are differentiated by their parameter lists. In the following example, the class Light now provides both an explicit implementation of the default constructor at (1) and a non-default constructor at (2). The constructors are overloaded, as is evident by their signatures. The non-default constructor is called when an object of class Light is created at (3), and the default constructor is likewise called at (4). Overloading of constructors allows appropriate initialization of objects on creation, depending on the constructor invoked.

```
class Light {
   // ...
   // Explicit Default Constructor
   Light() {                                    // (1)
 noOfWatts = 50;
      indicator = true;
      location  = "X";
   }

   // Non-default Constructor
   Light(int noOfWatts, boolean indicator, String location) { // (2)
 this.noOfWatts = noOfWatts;
      this.indicator = indicator;
      this.location  = location;
   }
   //...
}

class Greenhouse {
   // ...
   Light moreLight  = new Light(100, true, "Greenhouse");     // (3) OK.
Light firstLight = new Light();                              // (4) OK.
}
```

4.5 Scope Rules

Java provides explicit accessibility modifiers to control the accessibility of members in a class by external clients, but in two areas access is governed by specific scope rules:

- Class scope for members: how member declarations are accessed within the class.

- Block scope for local variables: how local variable declarations are accessed within a block.

Class Scope for Members

Class scope concerns accessing members (including inherited ones) from code within a class. Table 4.1 gives an overview of how static and non-static code in a class can access members of the class, including those that are inherited. Table 4.1 assumes the following declarations:

```
class SuperName {
   int instanceVarInSuper;
   static int staticVarInSuper;
```

```
    void instanceMethodInSuper()      { /* ... */ }
  static void staticMethodInSuper() { /* ... */ }     // ...
}

class ClassName extends SuperName {
    int instanceVar;
    static int staticVar;

    void instanceMethod()      { /* ... */ }
    static void staticMethod() { /* ... */ }
    // ...
}
```

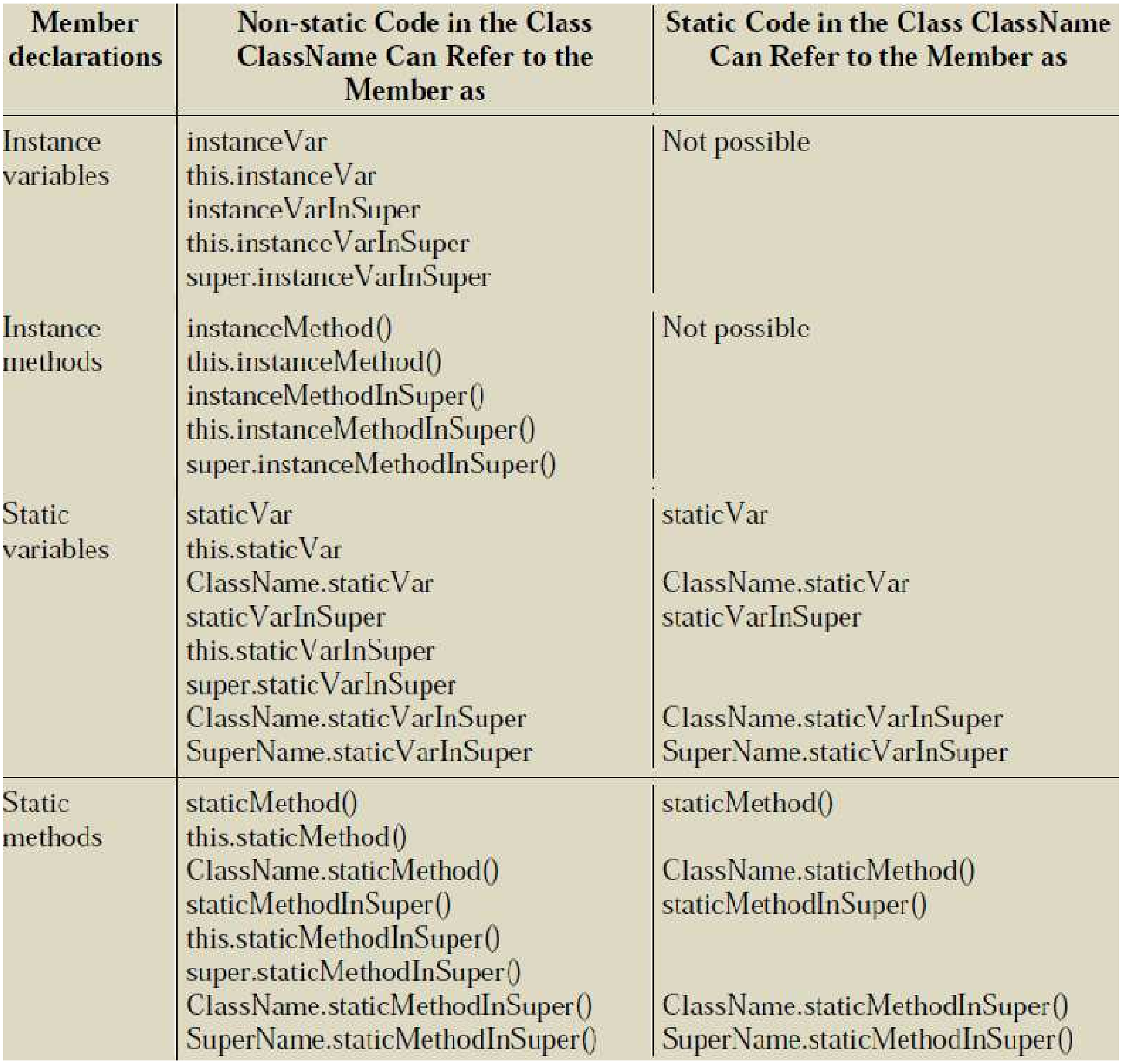
The golden rule is that static code cannot access non-static members by their simple names. Static code is not executed in the context of an object, therefore the references this and super are not available. An object has knowledge of its class, therefore, static members are always accessible in a non-static context.

Note that using the class name to access static members within the class is no different from how external clients access these static members.

Some factors that can influence the scope of a member declaration are

- shadowing of a field declaration, either by local variables or by declarations in the subclass
- initializers preceding the field declaration
- overriding an instance method from a superclass

Table 4.1. Accessing Members within a Class



Member declarations	Non-static Code in the Class ClassName Can Refer to the Member as	Static Code in the Class ClassName Can Refer to the Member as
Instance variables	instanceVar this.instanceVar instanceVarInSuper this.instanceVarInSuper super.instanceVarInSuper	Not possible

Instance methods	instanceMethod()
	this.instanceMethod()
	instanceMethodInSuper()
	this.instanceMethodInSuper()
	super.instanceMethodInSuper()
		Not possible

Static variables	staticVar this.staticVar ClassName.staticVar staticVarInSuper this.staticVarInSuper super.staticVarInSuper ClassName.staticVarInSuper SuperName.staticVarInSuper	staticVar ClassName.staticVar staticVarInSuper ClassName.staticVarInSuper SuperName.staticVarInSuper

Static methods	staticMethod()
	this.staticMethod()
	ClassName.staticMethod()
	staticMethodInSuper()
	this.staticMethodInSuper()
	super.staticMethodInSuper()
	ClassName.staticMethodInSuper()
	SuperName.staticMethodInSuper()
		staticMethod()
		ClassName.staticMethod()
		staticMethodInSuper()
		ClassName.staticMethodInSuper()
		SuperName.staticMethodInSuper()

Within a class C, reference variables of type C can be used to access all members in the class C, regardless of their accessibility modifiers. In Example 4.6, the method duplicateLight at (1) in class Light has a parameter oldLight and a local variable newLight that are references to Light objects. Even though the fields of the class are private, they are accessible through the two references (oldLight and newLight) in the method duplicateLight() as shown at (2), (3), and (4).

Example 4.6 Class Scope

```
class Light {
   // Instance variables
   private int     noOfWatts;       // wattage
   private boolean indicator;       // on or off
private String  location;        // placement
   // Instance methods
   public void switchOn()  { indicator = true; }
 public void switchOff() { indicator = false; }
public boolean isOn()   { return indicator; }



   public static Light duplicateLight(Light oldLight) {       // (1)
Light newLight = new Light();
      newLight.noOfWatts = oldLight.noOfWatts;              // (2)
 newLight.indicator = oldLight.indicator;              // (3)
newLight.location  = oldLight.location;               // (4)
 return newLight;
   }
}
```

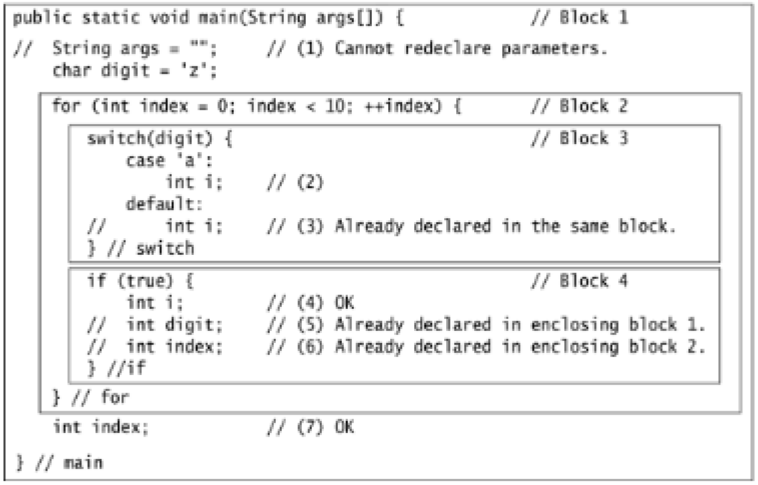
Block Scope for Local Variables

Declarations and statements can be grouped into a block using braces, {}. Blocks can be nested, and certain scope rules apply to local variable declarations in such blocks. A local declaration can appear anywhere in a block. The general rule is that a variable declared in a block is in scope inside the block in which it is declared, but it is not accessible outside of this block. It is not possible to redeclare a variable if a local variable of the same name is already declared in the current scope.

Local variables of a method are comprised of formal parameters of the method and variables that are declared in the method body. The local variables in a method are distinct for each invocation, and have their own storage.

A method body is a block. Parameters cannot be redeclared in the method body, as shown at (1) in Block 1

Figure 4.2. Block Scope



```
public static void main(String args[]) {            // Block 1
//  String args = "";     // (1) Cannot redeclare parameters.
    char digit = 'z';
    for (int index = 0; index < 10; ++index) {       // Block 2
        switch(digit) {                              // Block 3
            case 'a':
                int i;    // (2)
            default:
//              int i;    // (3) Already declared in the same block.
        } // switch
        if (true) {                                  // Block 4
            int i;        // (4) OK
//          int digit;    // (5) Already declared in enclosing block 1.
//          int index;    // (6) Already declared in enclosing block 2.
        } //if
    } // for
    int index;            // (7) OK
} // main
```

A local variable—already declared in an enclosing block and, therefore, visible in a nested block—cannot be redeclared in the nested block. These cases are shown at (3), (5), and (6).

A local variable in a block can be redeclared in another block if the blocks are disjoint, that is, they do not overlap. This is the case for variable i at (2) in Block 3 and at (4) in Block 4, as these two blocks are disjoint.

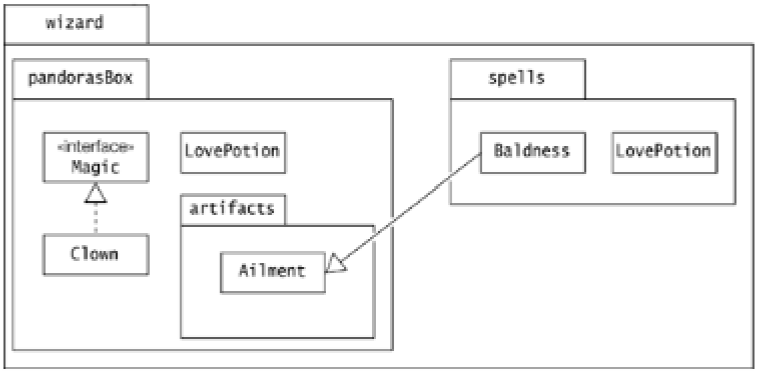
The scope of a declaration begins from where it is declared in the block and ends where this block terminates. The scope of the loop variable index is Block 2. Even though Block 2 is nested in Block 1, the declaration of the variable index at (7) in Block 1 is valid. Its scope spans from its declaration to the end of this block, and it does not overlap with that of the loop variable index in Block 2.

4.6 Packages

A package in Java is an encapsulation mechanism that can be used to group related classes, interfaces, and subpackages.

Figure 4.3 shows an example of a package hierarchy, comprising of a package called wizard that contains two other packages: pandorasBox and spells. The package pandorasBox has a class called Clown that implements an interface called Magic, also found in the same package. In addition, the package pandorasBox has a class called LovePotion and a subpackage called artifacts containing a class called Ailment. The package spells has two classes: Baldness and LovePotion. The class Baldness is a subclass of class Ailment found in the subpackage artifacts in the package pandorasBox.

Figure 4.3. Package Hierarchy



The dot (.) notation is used to uniquely identify package members in the package hierarchy. The class wizard.pandorasBox.LovePotion is different from the class wizard.spells.LovePotion. The Ailment class can be easily identified by the name wizard.pandorasBox.artifacts.Ailment. This is called the fully qualified name of the package member. It is not surprising that most Java programming environments map the fully qualified name of packages on the underlying (hierarchical) file system. For example, on a Unix system, the class file LovePotion.class corresponding to the class wizard.pandorasBox.LovePotion would be found under the directory wizard/pandorasBox.

A global naming scheme has been proposed to use the Internet domain names to uniquely identify packages. If the above package wizard was implemented by a company called Sorcerers Limited that owns the domain sorcerersltd.com, its fully qualified name would be:

com.sorcerersltd.wizard

The subpackage wizard.pandorasBox.artifacts could easily have been placed elsewhere, as long as it was uniquely identified. Subpackages do not affect the accessibility of the members. For all intent and purposes, subpackages are more an organizational feature rather than a language feature.

Defining Packages

A package hierarchy represents an organization of the Java classes and interfaces. It does not represent the source code organization of the classes and interfaces. The source code is of no consequence in this regard. Each Java source file (also called compilation unit) can contain zero or more definitions of classes and interfaces, but the compiler produces a separate class file containing the Java byte code for each of them. A class or interface can indicate that its Java byte code be placed in a particular package, using a package declaration.

The package statement has the following syntax: package <fully qualified package name>;

At most one package declaration can appear in a source file, and it must be the first statement in the unit. The package name is saved in the Java byte code for the types contained in the package.

Note that this scheme has two consequences. First, all the classes and interfaces in a source file will be placed in the same package. Secondly, several source files can be used to specify the contents of a package.

If a package declaration is omitted in a compilation unit, the Java byte code for the declarations in the compilation unit will belong to an unnamed package, which is typically synonymous with the current working directory on the host system.

Using Packages

The accessibility of types (classes and interfaces) in a package may deny access from outside the package. Given a reference type that is accessible from outside a package, the reference type can be accessed in two ways. The first form uses the fully qualified name of the type. However, writing long names can become tedious. The second form uses the import declaration to provide a shorthand notation for specifying the name of the type.

The import declarations must be the first statement after any package declaration in a source file. The simple form of the import declaration has the following syntax:

import <fully qualified type name>;

This is called single type import. As the name implies, such an import declaration provides a shorthand notation for a single class or interface. The simple name of the type (i.e., its identifier) can be used to access this particular type. Given the following import declaration:

import wizard.pandorasBox.Clown;

the name Clown can be used in the source file to refer to this class.

Alternatively, the following form of the import declaration can be used:

import <fully qualified package name>.*;

This is called type import on demand. It allows any type from the specified package to be accessed by its simple name.

An import declaration does not recursively import subpackages. The declaration does not result in inclusion of the source code of the types. The declaration only imports type names (i.e., it makes type names available to the code in a compilation unit).

All compilation units implicitly import the java.lang package. This is the reason why we can refer to the class String by its simple name, and not need to use its fully qualified name java.lang.String all the time.

Example 4.7 shows several usages of the import declaration. Here we will draw attention to the class Baldness in the file Baldness.java. This class relies on two classes that have the same simple name LovePotion but are in different packages: wizard.pandorasBox and wizard.spells, respectively. To distinguish between the two classes, we can use their fully qualified names. However, since one of them is in the same package as the class Baldness, it is enough to fully qualify the class from the other package. This solution is used in the following code. Such name conflicts can usually be resolved by using variations of the import declaration together with fully qualified names.

Example 4.7

```
// File: Baldness.java
package wizard.spells;                    // (1)Package declaration ...
import wizard.pandorasBox.artifacts.*;      // (3) Import from subpackage ...
public class Baldness extends Ailment {      // (4) Abbreviated name for Ailment
   wizard.pandorasBox.LovePotion tlcOne;    // (5) Fully qualified name     LovePotion tlcTwo;
// (6) Class in same package     ...
   }
}
...
```

The class Baldness extends the class Ailment, which is in the subpackage artifacts of the wizard.pandorasBox package. A new import declaration at (3) is used to import the types from the subpackage artifacts.

The following example shows how single type import can be used to disambiguate a class name when access to the class is ambiguous by simple name. The following import declaration allows the simple name List as shorthand for the java.awt.List class as expected:

```
import java.awt.*;          // imports all class names from java.awt
```

Given the following two import declarations:

```
import java.awt.*;          // imports all class names from java.awt
import java.util.*;         // imports all class names from java.util
```

the simple name List is now ambiguous as both the classes java.util.List and java.awt.List match.

Adding a single type import for the java.awt.List class explicitly allows the simple name List as a shorthand notation for this class:

```
 import java.awt.*;          // imports all class names from java.awt
import java.util.*;         // imports all class names from java.util
import java.awt.List;       // imports the class name List from java.awt
```

Compiling and Running Code from Packages

As mentioned earlier, a package hierarchy can be mapped on a hierarchical file system. We can think of a package name as a path in the file system. Referring to Example 4.7, the package name wizard.pandorasBox corresponds to the path name wizard/pandorasBox. The javac compiler can place the byte code in a directory that corresponds to the package declaration of the compilation unit. The Java byte code for all the classes (and interfaces) specified in the source files Clown.java and LovePotion.java will be placed in the directory named wizard/pandorasBox, as these source files have the following package declaration:

```
package wizard.pandorasBox;
```

The absolute path of the wizard/pandorasBox directory is specified by using the -d option (d for destination) when compiling with the javac compiler. Assuming that the current directory is called /pgjc/work, and all the source code files are to be found here, the command

```
>javac -d . Clown.java Ailment.java Baldness.java
```

issued in the work directory, will create ./wizard/pandorasBox (and any other subdirectories required) under the current directory, and place the Java byte code for all the classes (and interfaces) in the directories corresponding to the package names. The dot (.) after the -d option denotes the current directory. After compiling the code in Example 4.7 using the javac command above, the file hierarchy under the

/pgjc/work directory should mirror the package hierarchy in Figure 4.3. Without the -d option, the default behavior of the javac compiler is to place all class files directly under the current directory, rather than in the appropriate subdirectories.

How do we run the program? Since the current directory is /pgjc/work and we want to run Clown.class, the fully qualified name of the Clown class must be specified in the java command

```
>java wizard.pandorasBox.Clown
```

This will load the class Clown from the byte code in the file ./wizard/pandorasBox/Clown.class, and start the execution of its main() method.

The tools documentation for the Java 2 SDK explains how to organize packages in more elaborate schemes. In particular, the CLASSPATH environment variable can be used to specify multiple locations where Java tools should search when loading classes and resources.

4.7 Accessibility Modifiers for Top-level Classes and Interfaces

Top-level classes and interfaces within a package can be declared as public. This means that they are accessible from everywhere, both inside and outside of their package. If the accessibility modifier is omitted, then they are only accessible in the package and not in any other packages or subpackages. This is called package or default accessibility.

Example 4.7 Accessibility Modifiers for Classes and Interfaces // File: Clown.java

```
package wizard.pandorasBox;                    // (1) Package declaration
import wizard.pandorasBox.artifacts.Ailment;     // (2) Importing class
public class Clown implements Magic {
   LovePotion tlc;                             // (3) Class in same package
wizard.pandorasBox.artifacts.Ailment problem;// (4) Fully qualified class name
   Clown() {
      tlc = new LovePotion("passion");
      problem = new Ailment("flu");            // (5) Abbreviated class name
 }
   public void levitate()  { System.out.println("Levitating"); }
public void mixPotion() { System.out.println("Mixing " + tlc); }
  public void healAilment() { System.out.println("Healing " + problem); }
   public static void main(String[] args) {     // (6)
 Clown joker = new Clown();
      joker.levitate();
      joker.mixPotion();
      joker.healAilment();
   }
}

 interface Magic { void levitate(); }                                   // (7)
```

____ // File: LovePotion.java

```
package wizard.pandorasBox;              // (1) Package declaration

public class LovePotion {               // (2) Accessible outside package    String potionName;

   public LovePotion(String name) { potionName = name; }     public String toString() { return
potionName; } }
```

____ // File:

```
Ailment.java

package wizard.pandorasBox.artifacts;      // (1) Package declaration

public class Ailment {                    // (2) Accessible outside package     String ailmentName;

    public Ailment(String name) { ailmentName = name; }     public String toString() { return
ailmentName; } }

_______________________________________________________________ // File:
Baldness.java

package wizard.spells;                      // (1)Package declaration

import  wizard.pandorasBox.*;                                  //  (2)  Import  of  classes/interface  import
wizard.pandorasBox.artifacts.*;       // (3) Import from subpackage

public class Baldness extends Ailment {      // (4) Abbreviated name for

Ailment
    wizard.pandorasBox.LovePotion tlcOne;    // (5) Fully qualified name
LovePotion tlcTwo;                      // (6) Class in same package
Baldness(String name) {
        super(name);
        tlcOne = new wizard.pandorasBox.     // (7) Fully qualified name
LovePotion("romance");
        tlcTwo = new LovePotion();           // (8) Class in same package     }
}

class LovePotion // implements Magic         // (9) Not accessible

{ public void levitate(){} }
```

Compiling and running the program from the current working directory results in the following:

```
>javac -d . Clown.java Ailment.java Baldness.java
>java wizard.pandorasBox.Clown
Levitating
Mixing passion
Healing flu
```

In Example 4.7, the class Clown and the interface Magic are placed in a package called wizard.pandorasBox. The public class Clown is accessible from everywhere. The Magic interface has default accessibility, and can only be accessed within the package wizard.pandorasBox. It is not accessible from other packages, not even in any packages nested in this package.

The class LovePotion is also placed in the package called wizard.pandorasBox. The class has public accessibility and is, therefore, accessible from other packages. The two files Clown.java and LovePotion.java demonstrate how several compilation units can be used to group classes in the same package.

The class Clown, from the file Clown.java, uses the class Ailment. The example shows two ways in which a class can access classes from other packages:

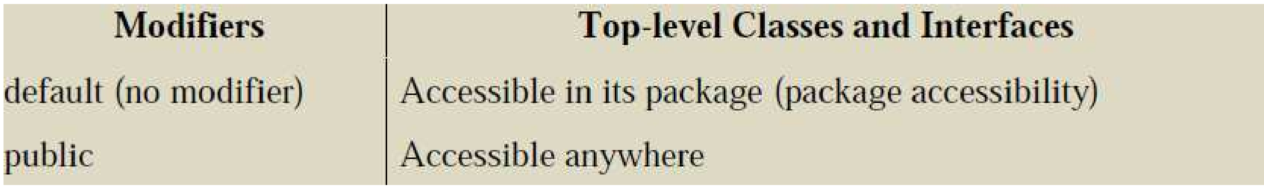
1. Denote the class by its fully qualified class name, as shown at (4) (wizard.pandorasBox.artifacts.Ailment).

2. Import the class explicitly from the package wizard.pandorasBox.artifacts as shown at (2) and use the simple class name Ailment as shown at (5).

In the file Baldness.java at (9), the class LovePotion wishes to implement the interface Magic from the package wizard.pandorasBox, but cannot do so, although the file imports from this package. The reason is that the interface Magic has default accessibility and can, therefore, only be accessed within the package wizard.pandorasBox.

Just because the class is accessible, it does not necessarily mean that members of the class are also accessible. Member accessibility is governed separately from class accessibility, as explained in Section 4.5

Table 4.2. Summary of Accessibility Modifiers for Classes and Interfaces



Modifiers	Top-level Classes and Interfaces
default (no modifier)	Accessible in its package (package accessibility)
public	Accessible anywhere

4.8 Other Modifiers for Classes

Modifiers abstract and final can be applied to top-level and nested classes.

abstract Classes

A class can be specified with the keyword abstract to indicate that it cannot be instantiated. A class might choose to do this if the abstraction it represents is so general that it needs to be specialized in order to be of practical use. A class Vehicle might be specified as abstract to represent the general abstraction of a vehicle, as creating instances of the class would not make much sense. Creating instances of non-abstract subclasses, like Car and Bus, would make more sense, as this would make the abstraction more concrete.

A class that has one or more abstract methods must be declared abstract. Obviously such classes cannot be instantiated, as their implementation is only partial. A class might choose this strategy to dictate certain behavior, but allow its subclasses the freedom to provide the relevant implementation. In other words, subclasses of the abstract class have to take a stand and provide implementations of inherited abstract methods before they can be instantiated. A subclass that does not provide an implementation of its inherited abstract methods must also be declared abstract.

In Example 4.8, the definition of abstract class Light has an abstract method kwhPrice at (1). This forces its subclasses to provide the implementation for this method. The subclass TubeLight provides an implementation for the method kwhPrice at (2). The class Factory creates an instance of class TubeLight at (3). Reference variables of an abstract class can be declared, as shown at (4), but an abstract class cannot be instantiated, as shown at (5). References of an abstract class can denote objects of the subclasses, as shown at (6).

Example 4.8 Abstract Classes

```
abstract class Light {
   // Fields
   int     noOfWatts;       // wattage
   boolean indicator;       // on or off
   String  location;        // placement

   // Instance methods
   public void switchOn()  { indicator = true; }
public void switchOff() { indicator = false; }
  public boolean isOn()   { return indicator; }
   // Abstract Instance Method
   abstract public double kwhPrice();              // (1) No method body
}
class TubeLight extends Light {
   // Fields
   int tubeLength;
   // Implementation of inherited abstract method.
 public double kwhPrice() { return 2.75; }        // (2)
}

public class Factory {
   public static void main(String[] args) {
      TubeLight cellarLight = new TubeLight();     // (3) OK
Light nightLight;                          // (4) OK
 //  Light tableLight = new Light();              // (5) Compile time error
  nightLight = cellarLight;                   // (6) OK
 System.out.println("KWH price: " + nightLight.kwhPrice());
}
}
Output from the program:
KWH price: 2.75
```

Interfaces just specify the method prototypes and not any implementation; they are, by their nature, implicitly abstract (i.e., they cannot be instantiated). Though it is legal, it is redundant to declare an interface with the keyword abstract.

final Classes

A class can be declared final to indicate that it cannot be extended; that is, one cannot declare subclasses of a final class. This implies that one cannot override any methods declared in such a class. In other words, the class behavior cannot be changed by subclassing. The class marks the lower boundary of its implementation inheritance hierarchy. Only a class whose definition is complete (i.e., has implementations of all its methods) can be declared final.

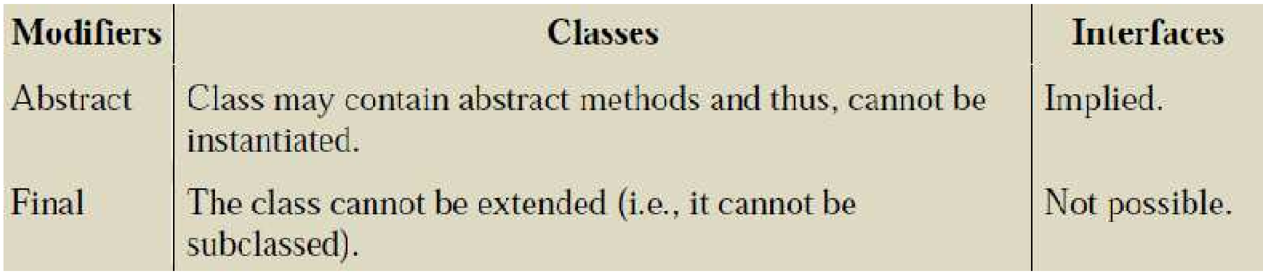
A final class must be complete, whereas an abstract class is considered incomplete. Classes, therefore, cannot be both final and abstract at the same time. Interfaces, which are inherently abstract, thus cannot be declared final. A final class and an interface represent two extremes when it comes to providing implementation. An abstract class represents a compromise between these two extremes.

The Java API includes many final classes; for example, java.lang.String which cannot be specialized any further by subclassing.

If it is decided that the class TubeLight in Example 4.8 cannot, or should not, be extended, it can be declared final:

```
final class TubeLight extends Light {
   // Fields
   int tubeLength;
   // Implementation of inherited abstract method.
 public double kwhPrice() { return 2.75; } }
```

Table 4.3. Summary of Other Modifiers for Classes and Interfaces



Modifiers	Classes	Interfaces
Abstract	Class may contain abstract methods and thus, cannot be instantiated.	Implied.
Final	The class cannot be extended (i.e., it cannot be subclassed).	Not possible.

4.9 Member Accessibility Modifiers

By specifying member accessibility modifiers, a class can control what information is accessible to clients (i.e., other classes). These modifiers help a class to define a contract so that clients know exactly what services are offered by the class.

Accessibility of members can be one of the following:

- public
- protected
- default (also called package accessibility)
- private

A member has package or default accessibility when no accessibility modifier is specified.

In the following discussion on accessibility modifiers for members of a class, keep in mind that the member accessibility modifier only has meaning if the class (or one of its subclasses) is accessible to the client. Also, note that only one accessibility modifier can be specified for a member. The discussion in this section applies to both instance and static members of top-level classes. In UML notation the prefixes + , #, and -, when applied to a member name, indicate public, protected, and private member accessibility, respectively. No prefix indicates default or package accessibility.

public Members

public accessibility is the least restrictive of all the accessibility modifiers. A public member is accessible from anywhere, both in the package containing its class and in other packages where this class is visible. This is true for both instance and static members.

Example 4.9 contains two source files, shown at (1) and (6). The package hierarchy defined by the source files is depicted in Figure 4.4, showing the two packages packageA and packageB containing their respective classes. Classes in package packageB use classes from package packageA. SuperclassA in packageA has two subclasses: SubclassA in packageA and SubclassB in packageB.

Example 4.9 Public Accessibility of Members

```
// Filename: SuperclassA.java                         (1)
package packageA;

public class SuperclassA {
   public int superclassVarA;                      // (2)
  public void superclassMethodA() {/*...*/}        // (3)
}
class SubclassA extends SuperclassA {
   void subclassMethodA() { superclassVarA = 10; }  // (4) OK.
 }
class AnyClassA {
   SuperclassA obj = new SuperclassA();
   void anyClassMethodA() {
      obj.superclassMethodA();                    // (5) OK.
}
}
 // Filename: SubclassB.java                          (6)
 package packageB;
import packageA.*;

public class SubclassB extends SuperclassA {
 void subclassMethodB() { superclassMethodA(); }  // (7) OK.
 }

class AnyClassB {
   SuperclassA obj = new SuperclassA();
   void anyClassMethodB() {
      obj.superclassVarA = 20;                    // (8) OK.
}
}
```

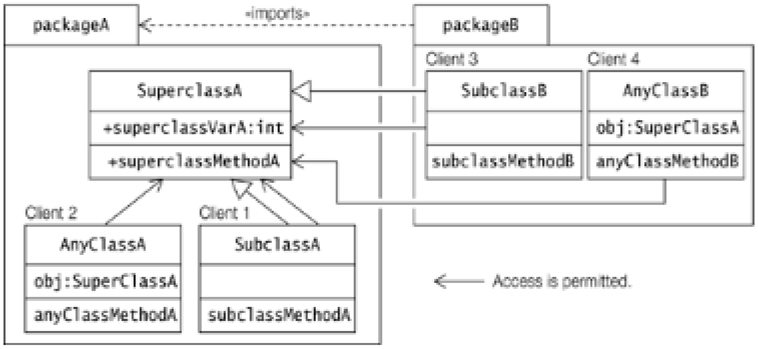


Figure 4.4. Public Accessibility

Accessibility is illustrated in Example 4.9 by the accessibility modifiers for the field superclassVarA and the method superclassMethodA at (2) and (3), respectively, defined in class SuperclassA. These members are accessed from four different clients in Example 4.9.

- Client 1: From a subclass in the same package, which accesses an inherited field. SubclassA at (4) is such a client.

- Client 2: From a non-subclass in the same package, which invokes a method on an instance of the class. AnyClassA at (5) is such a client.

- Client 3: From a subclass in another package, which invokes an inherited method. SubclassB at (7) is such a client.

- Client 4: From a non-subclass in another package, which accesses a field in an instance of the class. AnyClassB at (8) is such a client.

In Example 4.9, the field superclassVarA and the method superclass MethodA have public accessibility, and are accessible by all the four clients listed above. Subclasses can access their inherited public members by their simple name, and all clients can access public members through an instance of the class. Public accessibility is depicted in Figure 4.4.

protected Members

A protected member is accessible in all classes in the package containing its class, and by all subclasses of its class in any package where this class is visible. In other words, non-subclasses in other packages cannot access protected members from other packages. It is less restrictive than the default accessibility.

In Example 4.9, if the field superclassVarA and the method superclass MethodA have protected accessibility, then they are accessible within package packageA, and only accessible by subclasses in any other packages.

```
public class SuperclassA {
   protected int superclassVarA;                  // (2)
 protected void superclassMethodA() {/*...*/}     // (3)
}
```

Client 4 in package packageB cannot access these members, as shown in below.

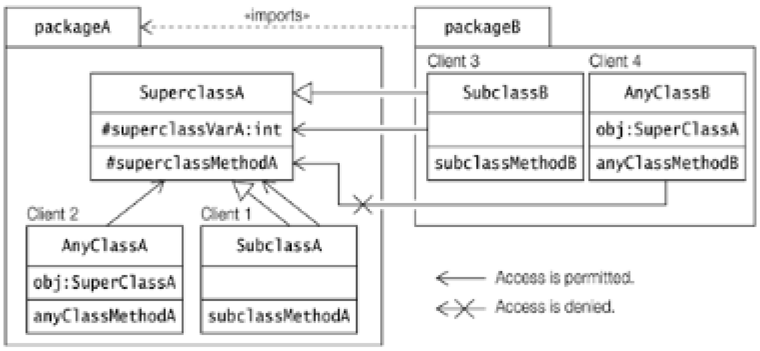


Figure 4.5. Protected Accessibility

A subclass in another package can only access protected members in the superclass via references of its own type or its subtypes. The following new definition of SubclassB in packageB from Example 4.9 illustrates the point: // Filename: SubclassB.java

```
package packageB;
import packageA.*;

public class SubclassB extends SuperclassA {         // In packageB.
    SubclassB objRefB = new SubclassB();             // (1)
    void subclassMethodB(SuperclassA objRefA) {
      objRefB.superclassMethodA();                 // (2) OK.
   objRefB.superclassVarA = 5;                   // (3) OK.
  objRefA.superclassMethodA();                  // (4) Not OK.
objRefA.superclassVarA = 10;                 // (5) Not OK.     }
}
```

The class SubclassB defines a field of type SubclassB (objRefB). The method subclassMethodB() has a formal parameter objRefA of type SuperclassA. Access is permitted to a protected member of the SuperclassA in packageA by a reference of the subclass, as shown at (2) and (3), but not by a reference of its superclass, as shown at (4) and (5). References to the field superclassVarA and the call to superclassMethodA() occur in SubclassB. These members are declared in SuperclassA. SubclassB is not involved in the implementation of a SuperclassA, which is the type of objRefA. Hence access to protected members at lines (4) and (5) is not permitted as these are not members of an object that can be guaranteed to be implemented by the code accessing them.

Accessibility to protected members of the superclass would be permitted via any references of subclasses of SubclassB. The above restriction helps to ensure that subclasses in packages different from their superclass can only access protected members of the superclass in their part of the implementation inheritance hierarchy.

Default Accessibility for Members

When no member accessibility modifier is specified, the member is only accessible by other classes in its class's package. Even if its class is visible in another (possibly nested) package, the member is not accessible there. Default member accessibility is more restrictive than protected member accessibility.

In Example 4.9, if the field superclassVarA and the method superclass MethodA are defined with no accessibility modifier, then they are only accessible within package packageA, but not in any other (possibly nested) packages.

```
 public class SuperclassA {
   int superclassVarA;                    // (2)
void superclassMethodA() {/*...*/}           // (3) }
```

The clients in package packageB (i.e. Clients 3 and 4) cannot access these members. This situation is depicted in Figure 4.6.

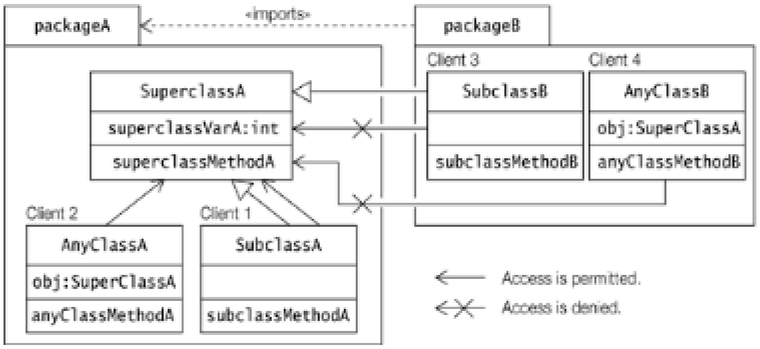


Figure 4.6. Default Accessibility

private Members

This is the most restrictive of all the accessibility modifiers. Private members are not accessible from any other class. This also applies to subclasses, whether they are in the same package or not. Since they are not accessible by simple name in a subclass, they are also not inherited by the subclass. This is not to be confused with the existence of such a member in the state of an object of the subclass. A standard design strategy is to make all fields private, and provide public accessor methods for them. Auxiliary methods are often declared private, as they do not concern any client.

In Example 4.9, if the field superclassVarA and the method superclass MethodA have private accessibility, then they are not accessible by any other clients.

```
public class SuperclassA {
   private int superclassVarA;                 // (2)
 private void superclassMethodA() {/*...*/}      // (3) }
```

None of the clients in Figure 4.7 can access these members.

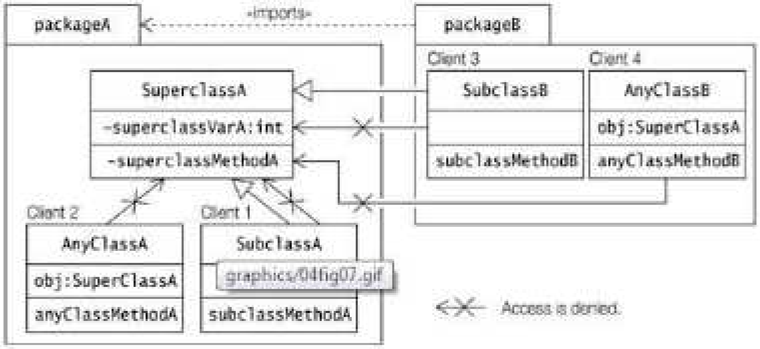
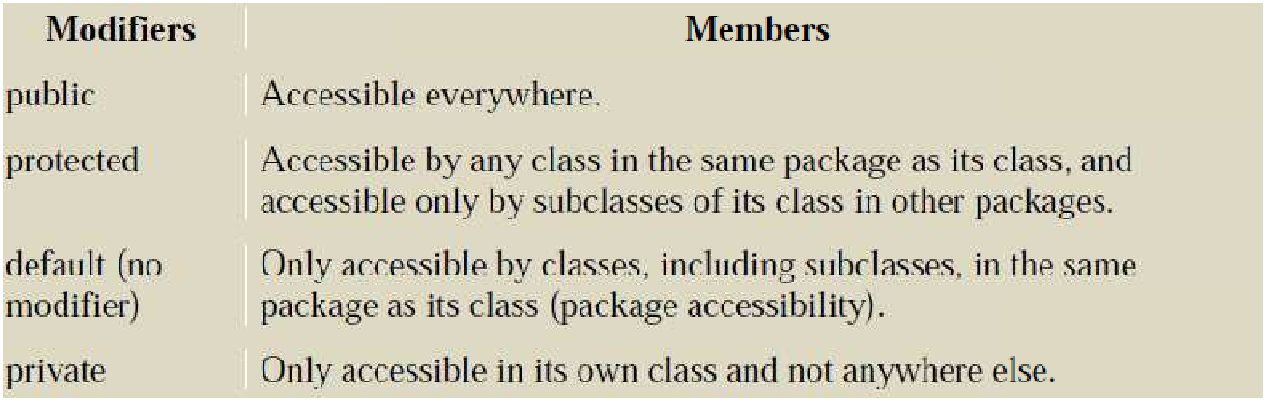


Figure 4.7. Private Accessibility

Table 4.4. Summary of Accessibility Modifiers for Members



Modifiers	Members
public	Accessible everywhere.
protected	Accessible by any class in the same package as its class, and accessible only by subclasses of its class in other packages.
default (no modifier)	Only accessible by classes, including subclasses, in the same package as its class (package accessibility).
private	Only accessible in its own class and not anywhere else.

4.10 Other Modifiers for Members

Certain characteristics of fields and/or methods can be specified in their declarations by the following keywords:

- static
- final
- abstract
- synchronized
- native
- transient
- volatile

static Members

The declaration of static members is prefixed by the keyword static to distinguish them from instance members.

Static members belong to the class in which they are declared, and are not part of any instance of the class. Depending on the accessibility modifiers of the static members in a class, clients can access these by using the class name, or through object references of the class. The class need not be instantiated to access its static members. Static variables (also called class variables) only exist in the class they are defined in. They are not instantiated when an instance of the class is created. In other words, the values of these variables are not a part of the state of any object. When the class is loaded, static variables are initialized to their default values if no explicit initialization expression is specified.

Static methods are also known as class methods. A static method in a class can directly access other static members in the class. It cannot access instance (i.e., non-static) members of the class, as there is no notion of an object associated with a static method. However, note that a static method in a class can always use a reference of the class's type to access its members, regardless of whether these members are static or not.

A typical static method might perform some task on behalf of the whole class and/or for objects of the class. In Example 4.10, the static variable counter keeps track of the number of instances of the Light class created. The example shows that the static method writeCount can only access static members directly, as shown at (2), but not non-static members, as shown at (3). The static variable counter will be initialized to the value 0 when the class is loaded at runtime. The main() method at (4) in class Warehouse shows how static members of class Light can be accessed using the class name, and via object references having the class type.

A summary of how static members are accessed in static and non-static code is given in Table 4.1.

Example 4.10 Accessing Static Members

```
class Light {
  // Fields
  int     noOfWatts;      // wattage
  boolean indicator;      // on or off
  String  location;       // placement

  // Static variable
  static int counter;     // No. of Light objects created.        (1)
  // Explicit Default Constructor
  Light() {
     noOfWatts = 50;
     indicator = true;

     location  = "X";

     ++counter;          // Increment counter.     }
```

```
    // Static method
    public static void writeCount() {
         System.out.println("Number of lights: " + counter);      // (2)
  // Compile error. Field noOfWatts is not accessible:
 // System.out.println("Number of Watts: " + noOfWatts);  // (3)     }
}
public class Warehouse {
    public static void main(String[] args) {                     // (4)
        Light.writeCount();                      // Invoked using class name
Light aLight = new Light();              // Create an object
 System.out.println(
            "Value of counter: " + Light.counter  // Accessed via class name
  );
        Light bLight = new Light();                  // Create another object          bLight.writeCount();
// Invoked using reference          Light cLight = new Light();                // Create another object
System.out.println(
            "Value of counter: " + cLight.counter // Accessed via reference         );

    }

}
```

Output from the program:

```
Number of lights: 0
Value of counter: 1
Number of lights: 2
Value of counter: 3
```

final Members

A final variable is a constant, despite being called a variable. Its value cannot be changed once it has been initialized. This applies to instance, static and local variables, including parameters that are declared final.

- A final variable of a primitive data type cannot change its value once it has been initialized.

- A final variable of a reference type cannot change its reference value once it has been initialized, but the state of the object it denotes can still be changed.

These variables are also known as blank final variables. Final static variables are commonly used to define manifest constants (also called named constants), for example Integer.MAX_VALUE, which is the maximum int value. Variables defined in an interface are implicitly final. Note that a final variable need not be initialized at its declaration, but it must be initialized once before it is

A final method in a class is complete (i.e., has an implementation) and cannot be overridden in any subclass. Subclasses are then restricted in changing the behavior of the method.

Final variables ensure that values cannot be changed, and final methods ensure that behavior cannot be changed.

The compiler is able to perform certain code optimizations for final members, because certain assumptions can be made about such members.

In Example 4.11, the class Light defines a final static variable at (1) and a final method at (2). An attempt to change the value of the final variable at (3) results in a compile-time error. The subclass TubeLight attempts to override the final method setWatts() from the superclass Light at (4), which is not permitted. The class Warehouse defines a final local reference aLight at (5). The state of the object denoted by aLight can be changed at (6), but its reference value cannot be changed as attempted at (7).

Example 4.11 Accessing Final Members

```
class Light {
   // Final static variable                (1)
   final public static double KWH_PRICE = 3.25;
   int noOfWatts;

   // Final instance method                (2)
   final public void setWatts(int watt) {
       noOfWatts = watt;
   }
   public void setKWH() {
     // KWH_PRICE = 4.10;             // (3) Not OK. Cannot be changed.     }
}




class TubeLight extends Light {
   // Final method in superclass cannot be overridden.
  // This method will not compile.
   /*
   public void setWatts(int watt) {       // (4) Attempt to override.
noOfWatts = 2*watt;
   }
   */

}
```

```
public class Warehouse {
   public static void main(String[] args) {

      final Light aLight = new Light();// (5) Final local variable.
aLight.noOfWatts = 100;          // (6) OK. Changing object state.
 //  aLight = new Light();           // (7) Not OK. Changing final reference.     }
}
```

abstract Methods

An abstract method has the following syntax:

abstract <accessibility modifier> <return type> <method name> (<parameter list>)
<throws clause>;

An abstract method does not have an implementation; that is, no method body is defined for an abstract method, only the method prototype is provided in the class definition. Its class is then abstract (i.e., incomplete) and must be explicitly declared as such. Subclasses of an abstract class must then provide the method implementation; otherwise, they are also abstract.

Only an instance method can be declared abstract. Since static methods cannot be overridden, declaring an abstract static method would make no sense. A final method cannot be abstract (i.e., cannot be incomplete) and vice versa. The keyword abstract cannot be combined with any nonaccessibility modifiers for methods. Methods specified in an interface are implicitly abstract, as only the method prototypes are defined in an interface.

synchronized Methods

Several threads can be executing in a program. They might try to execute several methods on the same object simultaneously. If it is desired that only one thread at a time can execute a method in the object, the methods can be declared synchronized. Their execution is then mutually exclusive among all threads. At any given time, at the most one thread can be executing a synchronized method on an object. This discussion also applies to static synchronized methods of a class.

In Example 4.12, both the push() and the pop() methods are synchronized in class StackImpl. Now, only one thread at a time can execute a synchronized method in an object of the class StackImpl. This means that it is not possible for the state of an object of the class StackImpl to be corrupted, for example, while one thread is pushing an element and another is popping the stack.

Example 4.12 Synchronized Methods

```
class StackImpl {
   private Object[] stackArray;
   private int topOfStack;
   // ...
   synchronized public void push(Object elem) { // (1)
stackArray[++topOfStack] = elem;
   }

   synchronized public Object pop() {          // (2)
Object obj = stackArray[topOfStack];
stackArray[topOfStack] = null;
      topOfStack--;
      return obj;
   }

   // Other methods, etc.

   public Object peek() { return stackArray[topOfStack]; } }
```

native Methods

Native methods are also called foreign methods. Their implementation is not defined in Java but in another programming language, for example, C or C++. Such a method can be declared as a member in a Java class definition. Since its implementation appears elsewhere, only the method prototype is specified in the class definition. The method prototype is prefixed with the keyword native. It can also specify checked exceptions in a throws clause, which cannot be checked by the compiler since the method is not implemented in Java. The next example shows how native methods are used.

The Java Native Interface (JNI) is a special API that allows Java methods to invoke native functions implemented in C.

In the following example, a native method in class Native is declared at (2). The class also uses a static initializer block at (1) to load the native library when the class is loaded. Clients of the Native class can call the native method like any another method, as at (3).

```
class Native {

 /*

    * The static block ensures that the native method library     * is loaded before the native
method is called.     */
```

```
   static {
      System.loadLibrary("NativeMethodLib");  // (1) Load native library.      }
    native void nativeMethod();                  // (2) Native method prototype.     // ...
   }
class Client {
   //...
   public static void main(String[] args) {
      Native aNative = new Native();
      aNative.nativeMethod();                  // (3) Native method call.     }
   //...
}
```

transient Fields

Objects can be stored using serialization. Serialization transforms objects into an output format that is conducive for storing objects. Objects can later be retrieved in the same state as when they were serialized, meaning that all fields included in the serialization will have the same values as at the time of serialization. Such objects are said to be persistent.

A field can be specified as transient in the class declaration, indicating that its value should not be saved when objects of the class are written to persistent storage. In the following example, the field currentTemperature is declared transient at (1), because the current temperature is most likely to have changed when the object is restored at a later date. However, the value of the field mass, declared at (2), is likely to remain unchanged. When objects of the class Experiment are serialized, the value of the field currentTemperature will not be saved, but that of the field mass will be as part of the state of the serialized object.

```
class Experiment implements Serializable {
 // ...
    // The value of currentTemperature will not persist
transient int currentTemperature;     // (1) Transient value.
 double mass;                          // (2) Persistent value. }
```

Specifying the transient modifier for static variables is redundant and, therefore, discouraged. Static variables are not part of the persistent state of a serialized object.

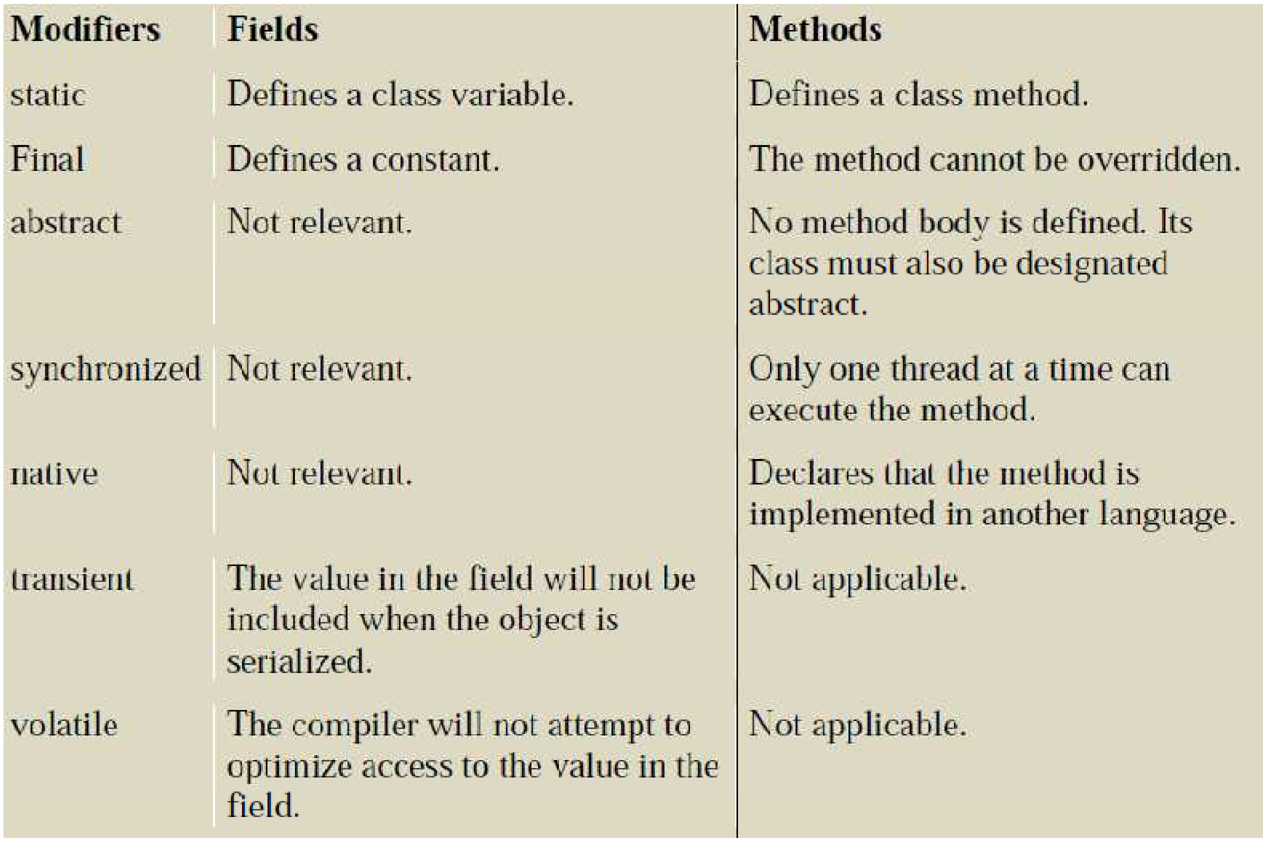
volatile Fields

During execution, compiled code might cache the values of fields for efficiency reasons. Since multiple threads can access the same field, it is vital that caching is not allowed to cause inconsistencies when reading and writing the value in the field. The volatile modifier can be used to inform the compiler that it should not attempt to perform optimizations on the field, which could cause unpredictable results when the field is accessed by multiple threads.

In the simple example that follows, the value of the field clockReading might be changed unexpectedly by another thread while one thread is performing a task that involves always using the current value of the field clockReading. Declaring the field as volatile ensures that a write operation will always be performed on the master field variable, and a read operation will always return the correct current value.

```
class VitalControl {
   // ...
   volatile long clockReading;
   // Two successive reads might give different results.  }
```

Table 4.5. Summary of Other Modifiers for Members



Modifiers	Fields	Methods
static	Defines a class variable.	Defines a class method.
Final	Defines a constant.	The method cannot be overridden.
abstract	Not relevant.	No method body is defined. Its class must also be designated abstract.
synchronized	Not relevant.	Only one thread at a time can execute the method.
native	Not relevant.	Declares that the method is implemented in another language.
transient	The value in the field will not be included when the object is serialized.	Not applicable.
volatile	The compiler will not attempt to optimize access to the value in the field.	Not applicable.

05

Control Flow, Exception Handling, Assertion

5.1 Overview of Control Flow Statements

Control flow statements govern the flow of control in a program during execution, that is, the order in which statements are executed in a running program. There are three main categories of control flow statements that are discussed in this chapter:

· Selection statements: if, if-else and switch.
· Iteration statements: while, do-while and for.
· Transfer statements: break, continue, return, try-catch-finally and assert.

5.2 Selection Statements

Java provides selection statements that allow the program to choose between alternative actions during execution. The choice is based on criteria specified in the selection statement. These selection statements are

· simple if Statement
• if-else Statement
• switch Statement

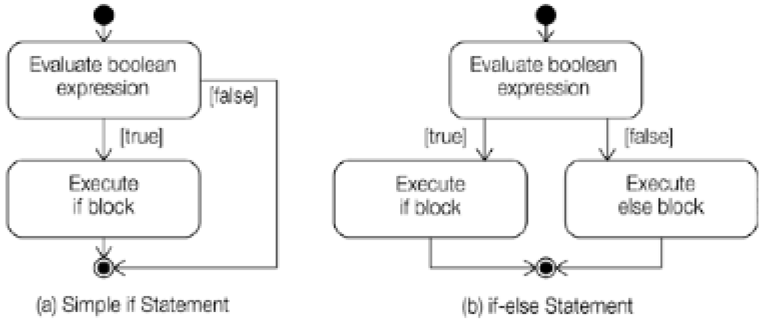
Simple if Statement

The simple if statement has the following syntax: if (<conditional expression>)
    <statement>

It is used to decide whether an action is to be performed or not, based on a condition. The condition is specified by <conditional expression> and the action to be performed is specified by <statement>.

The semantics of the simple if statement are straightforward. The <conditional expression> is evaluated first. If its value is true, then <statement> (called the if block) is executed and execution continues with the rest of the program. If the value is false, then the if block is skipped and execution continues with the rest of the program. The semantics are illustrated by the activity diagram in Figure 5.1.

Figure 5.1. Activity Diagram for if Statements



In the following examples of the if statement, it is assumed that the variables and the methods have been defined appropriately:

```
if (emergency)            // emergency is a boolean variable
 operate();

if (temperature > critical)
   soundAlarm();

if (isLeapYear() && endOfCentury())
   celebrate();

if (catIsAway()) {        // Block
   getFishingRod();
   goFishing();
}
```

Note that <statement> can be a block, and the block notation is necessary if more that one statement is to be executed when the <conditional expression> is true.

Since the <conditional expression> must be a boolean expression, it avoids a common programming error: using an expression of the form (a=b) as the condition, where inadvertently an assignment operator is used instead of a relational operator. The compiler will flag this as an error, unless both a and b are boolean.

Note that the if block can be any valid statement. In particular, it can be the empty statement (;) or the empty block ({}). A common programming error is an inadvertent use of the empty statement.

```
if (emergency); // Empty if block
   operate();  // Executed regardless of whether it was an emergency or not
.
```

if-else Statement

The if-else statement has the following syntax: if(<conditional expression>)

```
     <statement1>

 else

     <statement2>
```

It is used to decide between two actions, based on a condition.

The <conditional expression> is evaluated first. If its value is true, then <statement1> (the if block) is executed and execution continues with the rest of the program. If the value is false, then <statement2> (the else block) is executed and execution continues with the rest of the program. In other words, one of two mutually exclusive actions is performed. The else clause is optional; if omitted, the construct reduces to the simple if statement.

In the following examples of the if-else statement, it is assumed that all variables and methods have been defined appropriately:

```
if (emergency)
   operate();
else
   joinQueue();

if (temperature > critical)
   soundAlarm();
else
   businessAsUsual();

if (catIsAway()) {
   getFishingRod();
   goFishing();
} else
   playWithCat();
```

Since actions can be arbitrary statements, the if statements can be nested.

```
 if (temperature >= upperLimit) {          // (1)

if (danger)                          // (2) Simple if.

 soundAlarm();
   if (critical)                       // (3)
      evacuate();
```

```
    else                          // Goes with if at (3).
 turnHeaterOff();
} else                            // Goes with if at (1).
 turnHeaterOn();
```

The use of the block notation, {}, can be critical to the execution of if statements. The if statements (A) and (B) in the following examples do not have the same meaning.

The if statements (B) and (C) are the same, with extra indentation used in (C) to make the meaning evident. Leaving out the block notation in this case could have catastrophic consequences: the heater could be turned on when the temperature is above the upper limit.

```
// (A)
if (temperature > upperLimit) {        // (1) Block notation.
if (danger) soundAlarm();         // (2)
} else                          // Goes with if at (1).
 turnHeaterOn();

// (B)
if (temperature > upperLimit)          // (1) Without block notation.
if (danger) soundAlarm();         // (2)
else turnHeaterOn();              // Goes with if at (2).
// (C)
if (temperature > upperLimit)          // (1)
    if (danger)                    // (2)

        soundAlarm();

    else                          // Goes with if at (2).

 turnHeaterOn();
```

The rule for matching an else clause is that an else clause always refers to the nearest if that is not already associated with another else clause. Block notation and proper indentation can be used to make the meaning obvious.

Cascading if-else statements are a sequence of nested if-else statements where the if of the next if-else statement is joined to the else clause of the previous one. The decision to execute a block is then based on all the conditions evaluated so far.

```
if (temperature >= upperLimit) {                    // (1)
 soundAlarm();
```

```
   turnHeaterOff();
} else if (temperature < lowerLimit) {                    // (2)
soundAlarm();
   turnHeaterOn();
} else if (temperature == (upperLimit-lowerLimit)/2) {     // (3)
doingFine();
} else                                        // (4)
  noCauseToWorry();
```

The block corresponding to the first if condition that evaluates to true is executed, and the remaining ifs are skipped. In the example given above, the block at (3) will execute only if the conditions at (1) and (2) are false and the condition at (3) is true. If none of the conditions are true, the block associated with the last else clause is executed. If there is no last else clause, no actions are performed.

switch Statement

Conceptually the switch statement can be used to choose one among many alternative actions, based on the value of an expression. Its general form is as follows:

```
switch(<non-long integral expression>){
   case label1: <statement1>
   case label2: <statement2>
   ...
   case labeln: <statementn>
   default: <statement>
} // end switch
```

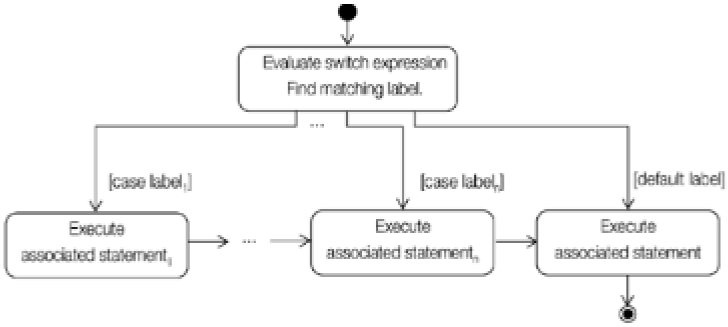
The syntax of the switch statement comprises a switch expression followed by the switch body, which is a block of statements. The type of the switch expression is non-long integral (i.e., char, byte, short, or int). The statements in the switch body can be labeled, defining entry points in the switch body where control can be transferred depending on the value of the switch expression. The semantics of the switch statement are as follows:

- The switch expression is evaluated first.
- The value of the switch expression is compared with the case labels. Control is transferred to the <statementi> associated with the case label that is equal to the value of the switch expression. After execution of the associated statement, control falls through to the next statement unless appropriate action is taken.

- If no case label is equal to the value of the switch expression, the statement associated with the default label is executed.

Figure 5.2 illustrates the flow of control through a switch statement.

Figure 5.2. Activity Diagram for switch Statement



All labels (including the default label) are optional and can be defined in any order in the switch body. There can be at most one default label in a switch statement. If it is left out and no valid case labels are found, the whole switch statement is skipped.

The case labels are constant expressions whose values must be unique, meaning no duplicate values are allowed. The case label values must be assignable to the type of the switch expression. In particular, the case label values must be in the range of the type of the switch expression. Note that the type of the case label cannot be boolean, long, or floating-point.

Example 5.1 Fall Through in switch Statement

```
public class Advice {
 public final static int LITTLE_ADVICE  = 0;
public final static int MORE_ADVICE    = 1;
public final static int LOTS_OF_ADVICE = 2;
    public static void main(String[] args) {
        dispenseAdvice(LOTS_OF_ADVICE);     }
  public static void dispenseAdvice(int howMuchAdvice) {
  switch(howMuchAdvice) {                            // (1)
case LOTS_OF_ADVICE:
                System.out.println("See no evil.");       // (2)
  case MORE_ADVICE:
                System.out.println("Speak no evil.");      // (3)
 case LITTLE_ADVICE:
                System.out.println("Hear no evil.");       // (4)                break;                                 //
(5)            default:

                System.out.println("No advice.");          // (6)         }

    }

}
```

Output from the program:

```
See no evil.
Speak no evil.
Hear no evil.
```

In Example 5.1, depending on the value of the howMuchAdvice parameter, different advice is printed in the switch statement at (1) in the method dispenseAdvice(). The example shows the output when the value of the howMuchAdvice parameter is LOTS_OF_ADVICE. In the switch statement, the associated statement at (2) is executed, giving one advice. Control then falls through to the statement at (3), giving the second advice. Control falls through to (4), dispensing the third advice, and finally executing the break statement at (5) causes control to exit the switch statement. Without the break statement at (5), control would continue to fall through the remaining statements if there were any. Execution of the break statement in a switch body transfers control out of the switch statement. If the parameter howMuchAdvice has the value MORE_ADVICE, then the advice at (3) and (4) is given. The value LITTLE_ADVICE results in only one advice at (4) being given. Any other value results in the default action, which announces that there is no advice.

The associated statement of a case label can be a list of statements (which need not be a statement block). The case label is prefixed to the first statement in each case. This is illustrated by the associated statement for the case label LITTLE_ADVICE in Example 5.1, which comprises statements (4) and (5). Example 5.2 makes use of a break statement inside a switch statement to convert a char value representing a digit to its corresponding word in English. Note that the break statement is the last statement in the list of statements associated with each case label. It is easy to think that the break statement is a part of the switch statement syntax, but technically it is not.

Example 5.2 Using break in switch Statement

```
public class Digits {
 public static void main(String[] args) {
     System.out.println(digitToString('7') + " " +        digitToString('8') + " " +
                  digitToString('6'));
   }
 public static String digitToString(char digit) {       String str = "";
     switch(digit) {
       case '1': str = "one";   break;
       case '2': str = "two";   break;
       case '3': str = "three"; break;
       case '4': str = "four";  break;
       case '5': str = "five";  break;
       case '6': str = "six";   break;

       case '7': str = "seven"; break;
```

```
            case '8': str = "eight"; break;
            case '9': str = "nine";  break;
            case '0': str = "zero";  break;
            default:  System.out.println(digit + " is not a digit!");         }
        return str;

    }

}
```

Output from the program:

```
seven eight six
```

Several case labels can prefix the same statement. They will all result in the associated statement being executed. This is illustrated in Example 5.3 for the switch statement at (1).

The first statement in the switch body must have a case label, or it is unreachable. This statement will never be executed since control can never be transferred to it. The compiler will flag this as an error.

Since each action associated with a case label can be an arbitrary statement, it can be another switch statement. In other words, switch statements can be nested. Since a

switch statement defines its own local block, the case labels in an inner block do not conflict with any case labels in an outer block. Labels can be redefined in nested blocks, unlike variables which cannot be redeclared in nested blocks. In Example 5.3, an inner switch statement is defined at (2). This allows further refinement of the action to take on the value of the switch expression, in cases where multiple labels are used in the outer switch statement. A break statement terminates the innermost switch statement in which it is executed.

Example 5.3 Nested switch Statement

```
public class Seasons {
 public static void main(String[] args) {
        int monthNumber = 11;
        switch(monthNumber) {                              // (1) Outer
     case 12: case 1: case 2:
                System.out.println("Snow in the winter.");
  break;
            case 3: case 4: case 5:
                System.out.println("Green grass in spring.");

break;

            case 6: case 7: case 8:
```

```
            System.out.println("Sunshine in the summer.");
break;
         case 9: case 10: case 11:                    // (2)
  switch(monthNumber) { // Nested switch            (3) Inner
  case 10:
                  System.out.println("Halloween.");
 break;
               case 11:
                  System.out.println("Thanksgiving.");
 break;
            } // end nested switch
            // Always printed for case labels 9, 10, 11
System.out.println("Yellow leaves in the fall."); // (4)
break;
         default:

            System.out.println(monthNumber + " is not a valid month.");       }

   }

}
```

Output from the program:

```
Thanksgiving.
Yellow leaves in the fall.
```

5.3 Iteration Statements

Loops allow a block of statements to be executed repeatedly (i.e., iterated). A boolean condition (called the loop condition) is commonly used to determine when to terminate the loop. The statements executed in the loop constitute the loop body. The loop body can be a single statement or a block. Java provides three language constructs for constructing loops:

- while statement
- do-while statement
- for statement

These loops differ in the order in which they execute the loop body and test the loop condition. The while and the for loops test the loop condition before executing the loop body, while the do-while loop tests the loop condition after execution of the loop body.

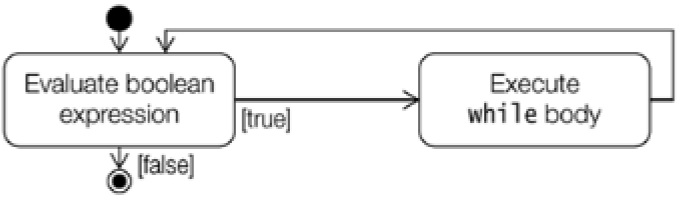
while Statement

The syntax of the while loop is

```
while (<loop condition>)
    <loop body>
```

The loop condition is evaluated before executing the loop body. The while statement executes the loop body as long as the loop condition is true. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop. If the loop condition is false to begin with, the loop body is not executed at all. In other words, a while loop can execute zero or more times. The loop condition must be a boolean expression. The flow of control in a while statement is shown in Figure 5.3.

Figure 5.3. Activity Diagram for while Statement



The while statement is normally used when the number of iterations is not known a priori.

```
while (noSignOfLife())
   keepLooking();
```

Since the loop body can be any valid statement, inadvertently terminating each line with the empty statement (;) can give unintended results.

```
while (noSignOfLife());     // Empty statement as loop body!
keepLooking();          // Statement not in the loop body.
```

do-while Statement

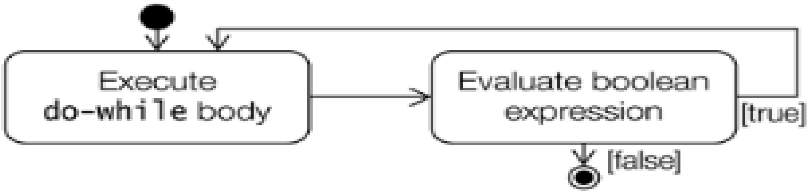
The syntax of the do-while loop is

```
do
   <loop body>
 while (<loop condition>);
```

The loop condition is evaluated after executing the loop body. The do-while statement executes the loop body until the loop condition becomes false. When the loop condition becomes false, the loop is terminated and execution continues with the statement immediately following the loop.

Note that the loop body is executed at least once. Figure 5.4 illustrates the flow of control in a do-while statement.

Figure 5.4. Activity Diagram for do-while Statement



The loop body in a do-while loop is invariably a statement block. It is instructive to compare the while and the do-while loops. In the examples below, the mice might never get to play if the cat is not away, as in the loop at (1). The mice do get to play at least once (at the peril of losing their life) in the loop at (2).

```
while (cat.isAway()) {       // (1)
   mice.play();
}

do {                         // (2)
   mice.play();
} while (cat.isAway());
```

for Statement

The for loop is the most general of all the loops. It is mostly used for counter-controlled loops, that is, when the number of iterations is known beforehand.

The syntax of the loop is as follows:

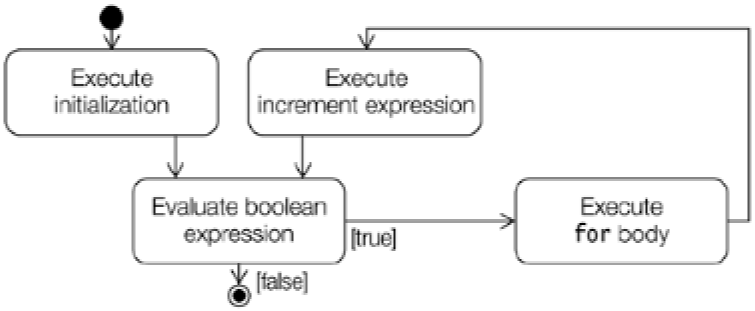
```
for (<initialization>; <loop condition>; <increment expression>)

<loop body>
```

The <initialization> usually declares and initializes a loop variable that controls the execution of the <loop body>. The <loop condition> is a boolean expression, usually involving the loop variable, such that if the loop condition is true, the loop body is executed; otherwise, execution continues with the statement following the for loop.

After each iteration (i.e., execution of the loop body), the <increment expression> is executed. This usually modifies the value of the loop variable to ensure eventual loop termination. The loop condition is then tested to determine if the loop body should be executed again. Note that the <initialization> is only executed once on entry to the loop. The semantics of the for loop are illustrated in Figure 5.5, and can be summarized by the following equivalent while loop code template:

```
 <initialization>
while (<loop condition>) {
   <loop body>
   <increment expression>
}
```

Figure 5.5. Activity Diagram for the for Statement



The following code creates an int array and sums the elements in the array.

```
  int sum = 0;
  int[] array = {12, 23, 5, 7, 19};
for (int index = 0; index < array.length; index++)   // (1)
sum += array[index];
```

The loop variable index is declared and initialized in the <initialization> section of the loop. It is incremented in the <increment expression> section. The for loop defines a local block such that the scope of this declaration is the for block, which comprises the <initialization>, the <loop condition>, the <loop body> and the <increment expression> sections.

The loop at (1) showed how a declaration statement can be specified in the <initialization> section. Such a declaration statement can also specify a comma-separated list of variables.

```
for (int i = 0, j = 1, k = 2; ... ; ...) ...;      // (2)
```

The variables i, j, and k in the declaration statement all have type int. All variables declared in the <initialization> section are local variables in the for block and obey the scope rules for local blocks. However, note that the following code will not compile, as variable declarations of different types (in this case, int and String) require declaration statements that are terminated by semicolons.

```
 for (int i = 0, String str = "@"; ... ; ...) ...;  // (3) Compile time error.
```

The <initialization> section can also be a comma-separated list of expression statements. For example, the loop at (2) can be rewritten by factoring out the variable declaration.

```
int i, j, k;  // Variable declaration

for (i = 0, j = 1, k = 2; ... ; ...) ...;      // (4) Just initialization
```

The <initialization> section is now a comma-separated list of three expressions. The expressions in such a list are always evaluated from left to right. Note that the variables i, j, and k are now obviously not local to the loop.

Declaration statements cannot be mixed with expression statements in the <initialization> section, as is the case at (5) in the following example. Factoring out the variable declaration, as at (6), leaves a legal comma-separated list of expression statements only.

```
// (5) Not legal and ugly.
for (int i = 0, System.out.println("This won't do!"); flag; i++) {
 // loop body
}

// (6) Legal, but still ugly.
int i;
for (i = 0, System.out.println("This is legal!"); flag; i++) {
// loop body
}
```

The <increment expression> can also be a comma-separated list of expression statements. The following code specifies a for loop that has a comma-separated list of three variables in the <initialization> section, and a comma-separated list of two expressions in the <increment expression> section.

```
// Legal usage but not recommended.
            int[][] sqMatrix = { {3, 4, 6}, {5, 7, 4}, {5, 8, 9} };
               for (int i = 0, j = sqMatrix[0].length - 1, asymDiagonal = 0;  // initialization
           i < sqMatrix.length;                                // loop condition
          i++, j--)                                  // increment expression
          asymDiagonal += sqMatrix[i][j];                  // loop body
```

All the sections in the for-header are optional. Any one of them can be left empty, but the two semicolons are mandatory. In particular, leaving out the <loop condition> signifies that the loop condition is true. The "crab", (;;), is commonly used to construct an infinite loop, where termination is presumably achieved through code in the loop body (see next section on transfer statements):

```
for (;;) Java.programming();       // Infinite loop
```

5.4 Transfer Statements

Java provides six language constructs for transferring control in a program:

- break
- continue
- return
- try-catch-finally
- throw
- assert

This section discusses the first three statements, and the remaining statements are discussed in subsequent sections.

Note that Java does not have a goto statement, although goto is a reserved word.

Labeled Statements

A statement may have a label.

<label> : <statement>

A label is any valid identifier and it always immediately proceeds the statement. Label names exist in their own name space, so that they do not conflict with names of packages, classes, interfaces, methods, fields, and local variables. The scope of a label is the statement prefixed by the label, meaning that it cannot be redeclared as a label inside the labeled statement—analogous to the scope of local variables.

```
   L1: if (i > 0) {
   L1: System.out.println(i);  // (1) Not OK. Label redeclared. }

L1: while (i < 0) {              // (2) OK.
   L2: System.out.println(i);
}

L1: {                            // (3) OK. Labeled block.     int j = 10;
   System.out.println(j);
}

L1: try {                        // (4) OK. Labeled try-catch-finally block.
  int j = 10, k = 0;
   L2: System.out.println(j/k);
} catch (ArithmeticException ae) {
   L3: ae.printStackTrace();
} finally {
   L4: System.out.println("Finally done.");
}
```

A statement can have multiple labels:

```
 LabelA: LabelB: System.out.println("Mutliple labels. Use judiciously.");
```

A declaration statement cannot have a label:

```
 L0: int i = 0;              // Compile time error.
```

A labeled statement is executed like it was non-labeled, unless it contains the break or continue statements. This is discussed in the next two subsections.

break Statement

The break statement comes in two forms: the unlabeled and the labeled form.

```
             break;          // the unlabeled form

             break <label>;    // the labeled form
```

The unlabeled break statement terminates loops (for, while, do-while) and switch statements which contain the break statement, and transfers control out of the current context (i.e., the closest enclosing block). The rest of the statement body is skipped, terminating the enclosing statement, with execution continuing after this statement.

In Example 5.4, the break statement at (1) is used to terminate a for loop. Control is transferred to (2) when the value of i is equal to 4 at (1), skipping the rest of the loop body and terminating the loop.

Example 5.4 also shows that the unlabeled break statement only terminates the innermost loop or switch statement that contains the break statement. The break statement at (3) terminates the inner for loop when j is equal to 2, and execution continues in the outer switch statement at (4) after the for loop.

Example 5.4 break Statement

```
class BreakOut {

   public static void main(String[] args) {

 for (int i = 1; i <= 5; ++i) {

         if (i == 4) break;        // (1) Terminate loop. Control to (2).
// Rest of loop body skipped when i gets the value 4.
  System.out.println(i + "\t" + Math.sqrt(i));
} // end for
     // (2) Continue here.



     int n = 2;
     switch (n) {
        case 1: System.out.println(n); break;
        case 2: System.out.println("Inner for loop: ");
               for (int j = 0; j < n; j++)
                if (j == 2)
                   break;   // (3) Terminate loop. Control to (4).
                     else
                   System.out.println(j);



        default: System.out.println("default: " + n); // (4) Continue here.        }

   }

}
```

Output from the program:

```
1    1.0
2    1.4142135623730951
3    1.7320508075688772
Inner for loop:
0
1
default: 2
```

A labeled break statement can be used to terminate any labeled statement that contains the break statement. Control is then transferred to the statement following the enclosing labeled statement. In the case of a labeled block, the rest of the block is skipped and execution continues with the statement following the block:

```
 out:

{          // (1) Labeled block

   // ...
```

```
    if (j == 10) break out; // (2) Terminate block. Control to (3).
System.out.println(j);  // Rest of the block not executed if j == 10.
 // ...
}
// (3) Continue here.
```

In Example 5.5, the program continues to add the elements below the diagonal of a square matrix until the sum is greater than 10. Two nested for loops are defined at (1) and (2). The outer loop is labeled outer at (1). The unlabeled break statement at (3) transfers control to (5) when it is executed, that is, it terminates the inner loop and control is transferred to the statement after the inner loop. The labeled break statement at (4) transfers control to (6) when it is executed (i.e., it terminates both the inner and the outer loop, transferring control to the statement after the loop labeled outer).

Example 5.5 Labeled break Statement

```
class LabeledBreakOut {
    public static void main(String[] args) {
        int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
          int sum = 0;
        outer:                                              // label
     for (int i = 0; i < squareMatrix.length; ++i){           // (1)
for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
 if (j == i) break;        // (3) Terminate this loop.
 //     Control to (5).
System.out.println("Element[" + i + ", " + j + "]: " +
 squareMatrix[i][j]);
sum += squareMatrix[i][j];
 if (sum > 10) break outer;// (4) Terminate both loops.
 //     Control to (6).              } // end inner loop

            // (5) Continue with outer loop.

        } // end outer loop

        // (6) Continue here.

        System.out.println("sum: " + sum);

    }

}
```

Output from the program:

```
Element[1, 0]: 2
Element[2, 0]: 9
sum: 11
```

continue Statement

Like the break statement, the continue statement also comes in two forms: the unlabeled and the labeled form.

```
continue;             // the unlabeled form
continue <label>;     // the labeled form
```

The continue statement can only be used in a for, while, or do-while loop to prematurely stop the current iteration of the loop body and proceed with the next iteration, if possible. In the case of the while and do-while loops, the rest of the loop body is skipped, that is, stopping the current iteration, with execution continuing with the <loop condition>. In the case of the for loop, the rest of the loop body is skipped, with execution continuing with the <increment expression>.

In Example 5.6, an unlabeled continue statement is used to skip an iteration in a for loop. Control is transferred to (2) when the value of i is equal to 4 at (1), skipping the rest of the loop body and continuing with the <increment expression> in the for statement.

Example 5.6 continue Statement

```
class Skip {
    public static void main(String[] args) {
        for (int i = 1; i <= 5; ++i) {
            if (i == 4) continue;             // (1) Control to (2).
  // Rest of loop body skipped when i has the value 4.
System.out.println(i + "\t" + Math.sqrt(i));
 // (2). Continue with increment expression.
 } // end for
    }
}
```

Output from the program:

```
1       1.0
2       1.4142135623730951
3       1.7320508075688772
5       2.23606797749979
```

A labeled continue statement must occur within a labeled loop that has the same label. Execution of the labeled continue statement then transfers control to the end of that enclosing labeled loop. In Example 5.7, the unlabeled continue statement at (3) transfers control to (5) when it is executed; that is, the rest of the loop body is skipped and execution continues with the next iteration of the inner loop. The labeled continue statement at (4) transfers control to (6) when it is executed (i.e., it terminates the inner loop but execution continues with the next iteration of the loop labeled outer). It is instructive to compare the output from Example 5.5 (labeled break) and Example 5.7 (labeled continue).

Example 5.7 Labeled continue Statement

```
class LabeledSkip {
    public static void main(String[] args) {
        int[][] squareMatrix = {{4, 3, 5}, {2, 1, 6}, {9, 7, 8}};
            int sum = 0;
        outer:                                          // label
          for (int i = 0; i < squareMatrix.length; ++i){           // (1)
       for (int j = 0; j < squareMatrix[i].length; ++j) {  // (2)
        if (j == i) continue;                   // (3) Control to (5).
      System.out.println("Element[" + i + ", " + j + "]: " +
squareMatrix[i][j]);
  sum += squareMatrix[i][j];
                if (sum > 10) continue outer;           // (4) Control to (6).
 // (5) Continue with inner loop.
            } // end inner loop
            // (6) Continue with outer loop.
        } // end outer loop
        System.out.println("sum: " + sum);
    }
}
```

Output from the program:

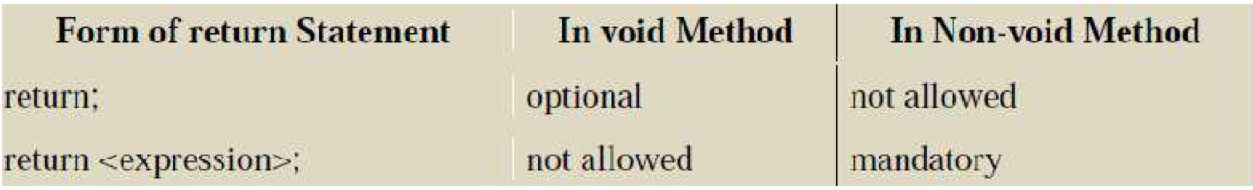
```
Element[0, 1]: 3
Element[0, 2]: 5
Element[1, 0]: 2
Element[1, 2]: 6
Element[2, 0]: 9
sum: 25
```

return Statement

The return statement is used to stop execution of a method and transfer control back to the calling code (a.k.a. the caller). The usage of the two forms of the return statement is dictated by whether it is used in a void or a non-void method. The first form does not return any value to the calling code, but the second form does. Note that the keyword void does not represent any type.

The <expression> must evaluate to a primitive value or a reference value, and its type must be assignable to the return type in the method prototype. As can be seen from Table 5.1, non-void methods must specify a return value using the return statement. A void method need not have a return statement —in which case control normally returns to the caller after the last statement in the method's body has been executed. The first form of the return statement can also be used in constructors, as these also do not return a value. Example 5.8 illustrates the usage of the return statement summarized in Table 5.1.

Table 5.1. return Statement



Form of return Statement	In void Method	In Non-void Method
return;	optional	not allowed
return <expression>;	not allowed	mandatory

Example 5.8 The return Statement

```
public class ReturnDemo {
   public static void main (String[] args) { // (1) void method can use return.
    if (args.length == 0) return;
      output(checkValue(args.length));
   }
   static void output(int value) { // (2) void method need not use return.
System.out.println(value);
      return 'a';                // Not OK. Cannot return a value.
   }
   static int checkValue(int i) {  // (3) non-void method must return a value.        if (i > 3)
        return i;                // OK.
      else
        return 2.0;              // Not OK. double not assignable to int.     }
}
```

5.5 Stack-based Execution and Exception Propagation

An exception in Java is a signal that indicates the occurrence of some important or unexpected condition during execution. For example, a requested file cannot be found, or an array index is out of bounds, or a network link failed. Explicit checks in the code for such conditions can easily result in incomprehensible code. Java provides an exception handling mechanism for systematically dealing with such error conditions.

The exception mechanism is built around the throw-and-catch paradigm. To throw an exception is to signal that an unexpected error condition has occurred. To catch an exception is to take appropriate action to deal with the exception. An exception is caught by an exception handler, and the exception need not be caught in the same context that it was thrown in. The runtime behavior of the program determines which exceptions are thrown and how they are caught. The throw-and-catch principle is embedded in the try-catch-finally construct.

Several threads can be executing in the JVM. Each thread has its own runtime stack (also called the call stack or the invocation stack) that is used to handle execution of methods. Each element on the stack (called an activation record or a stack frame) corresponds to a method call. Each new call results in a new activation record being pushed on the stack, which stores all the pertinent information such as storage for the local variables. The method with the activation record on top of the stack is the one currently executing. When this method finishes executing, its record is popped from the stack. Execution then continues in the method corresponding to the activation record which is now uncovered on top of the stack. The methods on the stack are said to be active, as their execution has not completed. At any given time, the active methods on a runtime stack comprise what is called the stack trace of a thread's execution.

Example 5.9 is a simple program to illustrate method execution. It calculates the average for a list of integers, given the sum of all the integers and the number of integers. It uses three methods:

- The method main() which calls the method printAverage() with parameters giving the total sum of the integers and the total number of integers, (1).
- The method printAverage() in its turn calls the method computeAverage(), (3).
- The method computeAverage() uses integer division to calculate the average and returns the result, (7).

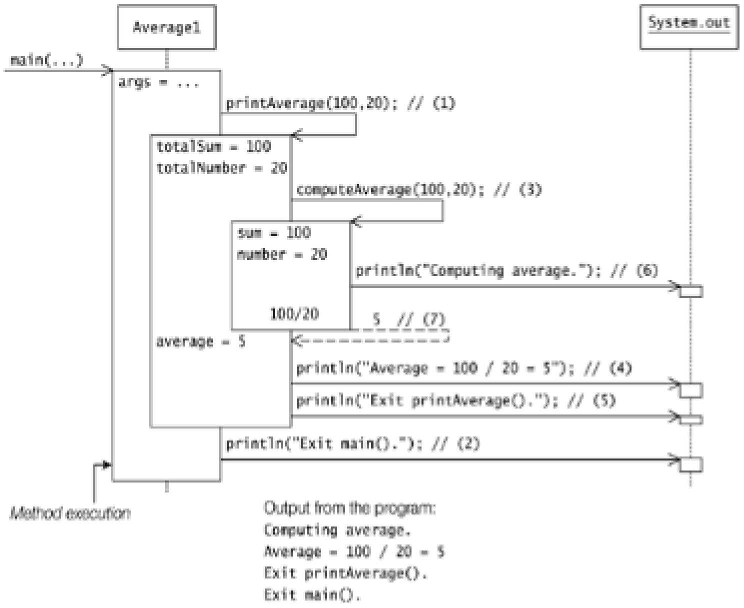
Example 5.9 Method Execution

```
public class Average1 {

    public static void main(String[] args) {
        printAverage(100,0);                              // (1)
System.out.println("Exit main().");                   // (2)     }

    public static void printAverage(int totalSum, int totalNumber) {
 int average = computeAverage(totalSum, totalNumber);  // (3)
  System.out.println("Average = " +                    // (4)
  totalSum + " / " + totalNumber + " = " + average);
   System.out.println("Exit printAverage().");          // (5)     }
   public static int computeAverage(int sum, int number) {
System.out.println("Computing average.");            // (6)
return sum/number;                                   // (7)     }
}
```

Output of program execution:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Execution of Example 5.9 is illustrated in Figure 5.6. Each method execution is shown as a box with the local variables. The box height indicates how long a method is active. Before the call to the method System.out.println() at (6) in Figure 5.6, the stack trace comprises of the three active methods: main(), printAverage() and computeAverage(). The result 5 from the method computeAverage() is returned at (7) in Figure 5.6. The output from the program is in correspondence with the sequence of method calls in Figure 5.6.


Average1
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
computeAverage(100,20); // (3)
sum = 100
number = 20
println("Computing average."); // (6)
100/20
5 // (7)
average = 5
println("Average = 100 / 20 = 5"); // (4)
println("Exit printAverage()."); // (5)
println("Exit main()."); // (2)
Method execution

Output from the program:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Figure 5.6. Method Execution

If the method call at (1) in Example 5.9

```
printAverage(100, 20);                   // (1)
```

is replaced with

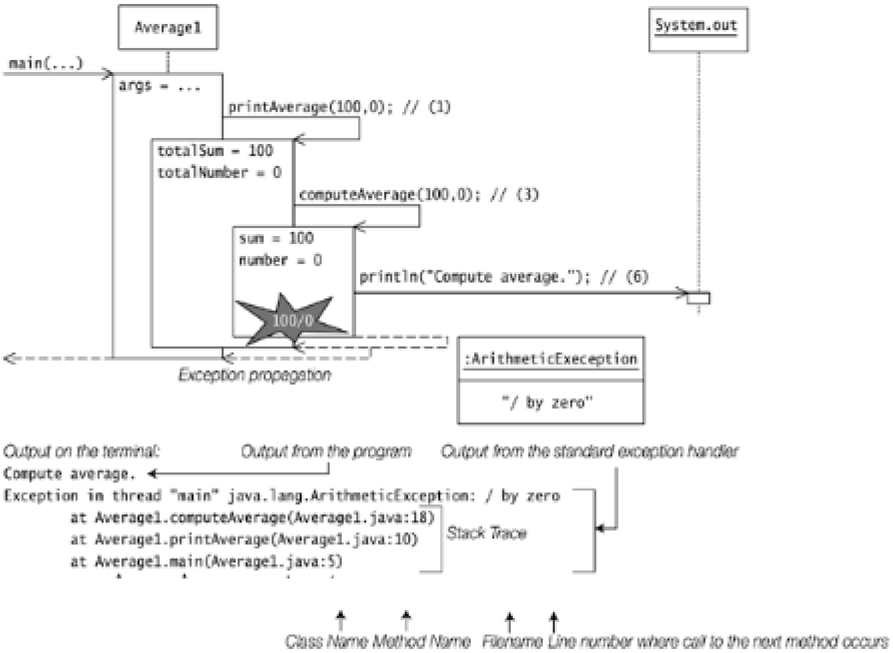
```
printAverage(100,  0);                   // (1)
```

and the program is run again, the output is as follows:

```
Computing average.
Exception in thread "main" java.lang.ArithmeticException: / by zero

at Average1.computeAverage(Average1.java:18)
 at Average1.printAverage(Average1.java:10)
  at Average1.main(Average1.java:5)
```

Figure 5.7 illustrates the program execution. All goes well until the return statement at (7) in the method computeAverage() is executed. An error condition occurs in calculating the expression sum/number, because integer division by 0 is an illegal operation. This error condition is signalled by the JVM by throwing an ArithmeticException. This exception is propagated by the JVM through the runtime stack as explained on the next page.

Figure 5.7. Exception Propagation



```
Compute average.
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Average1.computeAverage(Average1.java:18)
        at Average1.printAverage(Average1.java:10)
        at Average1.main(Average1.java:5)
```

Figure 5.7 illustrates the case where an exception is thrown and the program does not take any explicit action to deal with the exception. In Figure 5.7, execution of the computeAverage() method is stopped at the point where the exception is thrown. The execution of the return statement at (7) never gets completed. Since this method does not have any code to deal with the exception, its execution is likewise terminated abruptly and its activation record popped. We say that the method completes abruptly. The exception is then offered to the method whose activation is now on top of the stack (method printAverage()). This method does not have any code to deal with the exception either, so its execution completes abruptly. Lines (4) and (5) in the method printAverage() never get executed. The exception now propagates to the last active method (method main()). This does not deal with the exception either. The main() method also completes abruptly. Line (2) in the main() method never gets executed. Since the exception is not caught by any of the active methods, it is dealt with by the main thread's default exception handler. The default exception handler usually prints the name of the exception, with an explanatory message, followed by a printout of the stack trace at the time the exception was thrown. An uncaught exception results in the death of the thread in which the exception occurred.

If an exception is thrown during the evaluation of the left-hand operand of a binary expression, then the right operand is not evaluated. Similarly if an exception is thrown during the evaluation of a list of expressions (for example, a list of actual parameters in a method call), then evaluation of the rest of the list is skipped.

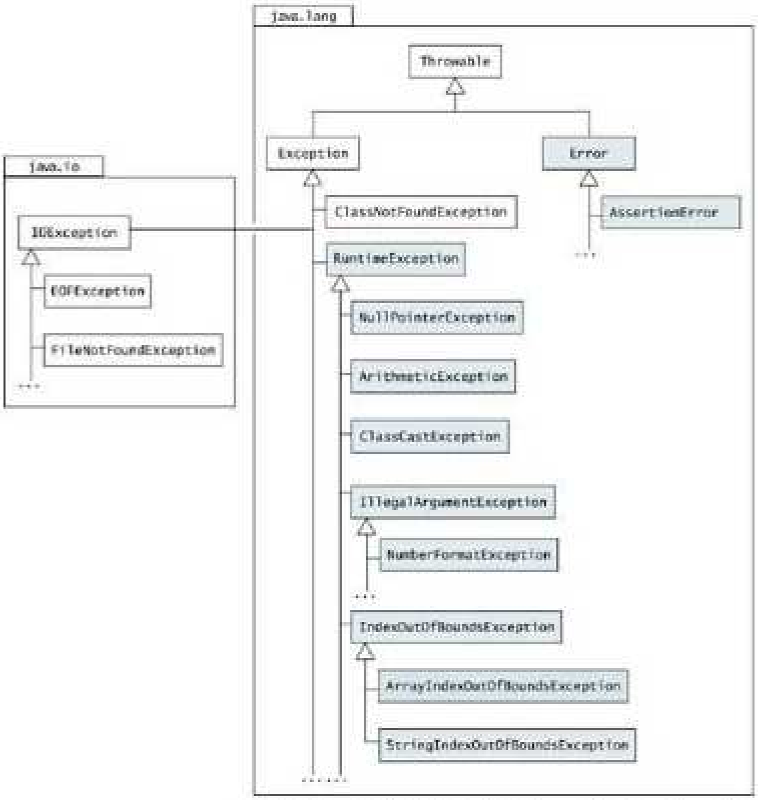
If the line numbers in the stack trace are not printed in the output as shown previously, it is advisable to turn off the JIT (Just-in-Time) compilation feature of the JVM in the Java 2 SDK:

```
>java -Djava.compiler=NONE Average1
```

5.6 Exception Types

Exceptions in Java are objects. All exceptions are derived from the java.lang. Throwable class. Figure 5.8 shows a partial hierarchy of classes derived from the Throwable class. The two main subclasses Exception and Error constitute the main categories of throwables, the term used to refer to both exceptions and errors. Figure 5.8 also shows that not all exception classes are found in the same package.

Figure 5.8. Partial Exception Inheritance Hierarchy



Classes that are shaded (and their subclasses) represent unchecked exceptions.

The Throwable class provides a String variable that can be set by the subclasses to provide a detail message. The purpose of the detail message is to provide more information about the actual exception. All classes of throwables define a one-parameter constructor that takes a string as the detail message.

The class Throwable provides the following common methods to query an exeception: String getMessage()

Returns the detail message.

void printStackTrace()

Prints the stack trace on the standard error stream. The stack trace comprises the method invocation sequence on the runtime stack when the exception was thrown. The stack trace can also be written to a PrintStream or a PrintWriter by supplying such a destination as an argument to one of the two overloaded printStackTrace() methods.

String toString()

Returns a short description of the exception, which typically comprises the class name of the exception together with the string returned by the getMessage() method.

Class Exception

The class Exception represents exceptions that a program would want to be made aware of during execution. Its subclass RuntimeException represents many common programming errors that manifest at runtime (see the next subsection). Other subclasses of the Exception class define other categories of exceptions, for example, I/O-related exceptions (IOException, FileNotFoundException, EOFException) and GUI-related exceptions (AWTException).

Class RuntimeException

Runtime exceptions, like out-of-bound array indices (ArrayIndexOutOfBounds Exception), uninitialized references (NullPointerException), illegal casting of references (ClassCastException), illegal parameters (IllegalArgumentException), division by zero (ArithmeticException), and number format problems (NumberFormatException) are all subclasses of the java.lang.RuntimeException class, which is a subclass of the Exception class. As these runtime exceptions are usually caused by program bugs that should not occur in the first place, it is more appropriate to treat them as faults in the program design, rather than merely catching them during program execution.

Class Error

The subclass AssertionError of the java.lang.Error class is used by the Java assertion facility. Other subclasses of the java.lang.Error class define exceptions that indicate class linkage (LinkageError), thread (ThreadDeath), and virtual machine (VirtualMachineError) related problems. These are invariably never explicitly caught and are usually irrecoverable.

Checked and Unchecked Exceptions

Except for RuntimeException, Error, and their subclasses, all exceptions are called checked exceptions. The compiler ensures that if a method can throw a checked exception, directly or indirectly, then the method must explicitly deal with it. The method must either catch the exception and take the appropriate action, or pass the exception on to its caller.

Exceptions defined by Error and RuntimeException classes and their subclasses are known as unchecked exceptions, meaning that a method is not obliged to deal with these kinds of exceptions. They are either irrecoverable (exemplified by the Error class) and the program should not attempt to deal with them, or they are programming errors (examplified by the RuntimeException class) and should be dealt with as such and not as exceptions.

Defining New Exceptions

New exceptions are usually defined to provide fine-grained categorization of exceptional conditions, instead of using existing exception classes with descriptive detail messages to differentiate between the conditions. New exceptions usually extend the Exception class directly or one of its checked subclasses, thereby making the new exceptions checked.

As exceptions are defined by classes, they can declare fields and methods, thus providing more information as to their cause and remedy when they are thrown and caught. The super() call can be used to set a detail message in the throwable. Note that the exception class must be instantiated to create an exception object that can be thrown and subsequently caught and dealt with. The code below sketches a class definition for an exception that can include all pertinent information about the exception.

```
public class EvacuateException extends Exception {
  // Data
    Date date;
    Zone zone;
    TransportMode transport;

    // Constructor
    public EvacuateException(Date d, Zone z, TransportMode t) {
   // Call the constructor of the superclass
 super("Evacuation of zone " + z);
```

```
        // ...
    }
    // Methods
    // ...
}
```

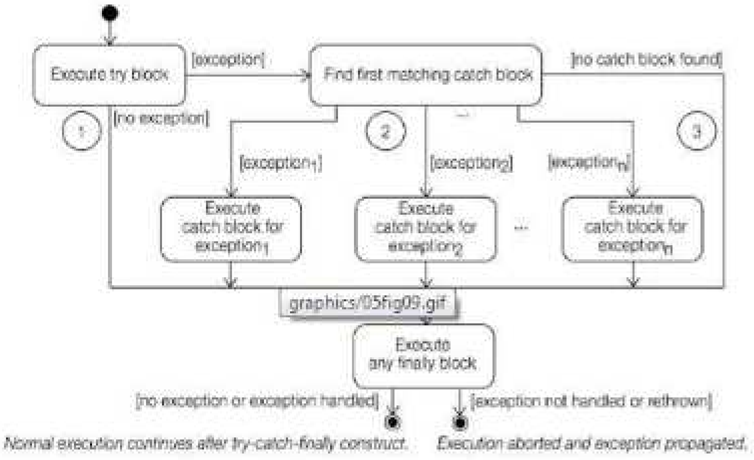
5.7 Exception Handling: try, catch, and finally

The mechanism for handling execeptions is embedded in the try-catch-finally construct, which has the following general form:

```
try {                                   // try block
    <statements>
} catch (<exception type1> <parameter1>) {  // catch block
<statements>
}
...
  catch (<exception typen> <parametern>) {  // catch block
 <statements>
} finally {                             // finally block
<statements>
}
```

Exceptions thrown during execution of the try block can be caught and handled in a catch block. A finally block is guaranteed to be executed, regardless of the cause of exit from the try block, or whether any catch block was executed. Figure 5.9 shows three typical scenarios of control flow through the try-catch-finally construct.

Figure 5.9. try-catch-finally Block


Execute try block
[exception]
Find first matching catch block
[no catch block found]
[no exception]
1
2
3
[exception$_1$]
[exception$_2$]
[exception$_n$]
Execute catch block for exception$_1$
Execute catch block for exception$_2$
...
Execute catch block for exception$_n$
graphics/05fig09.gif
Execute any finally block
[no exception or exception handled]
[exception not handled or rethrown]
Normal execution continues after try-catch-finally construct.
Execution aborted and exception propagated.

A few aspects about the syntax of this construct should be noted. The block notation is mandatory. For each try block there can be zero or more catch blocks, but only one finally block. The catch blocks and finally block must always appear in conjunction with a try block, and in the above order. A try block must be followed by either at least one catch block or one finally block. Each catch block defines an exception handler. The header of the catch block takes exactly one argument, which is the exception its block is willing to handle. The exception must be of the Throwable class or one of its subclasses.

Each block (try, catch, or finally) of a try-catch-finally construct can contain arbitrary code, which means that a try-catch-finally construct can also be nested in any such block. However, such nesting can easily make the code difficult to read and is best avoided.

try Block

The try block establishes a context that wants its termination to be handled. Termination occurs as a result of encountering an exception, or from successful execution of the code in the try block.

For all exits from the try block, except those due to exceptions, the catch blocks are skipped and control is transferred to the finally block, if one is specified. For all exits from the try block resulting from exceptions, control is transferred to the catch blocks—if any such blocks are specified—to find a matching catch block ((2) in Figure 5.9). If no catch block matches the thrown exception, control is transferred to the finally block, if one is specified.

catch Block

Only an exit from a try block resulting from an exception can transfer control to a catch block. A catch block can only catch the thrown exception if the exception is assignable to the parameter in the catch block. The code of the first such catch block is executed and all other catch blocks are ignored.

On exit from a catch block, normal execution continues unless there is any pending exception that has been thrown and not handled. If this is the case, the method is aborted and the exception is propagated up the runtime stack as explained earlier.

After a catch block has been executed, control is always transferred to the finally block, if one is specified. This is always true as long as there is a finally block, regardless of whether the catch block itself throws an exception.

In Example 5.10, the method printAverage() calls the method computeAverage() in a try-catch construct at (4). The catch block is declared to catch exceptions of type ArithmeticException. The catch block handles the exception by printing the stack trace and some additional information at (7) and (8), respectively. Execution of the program is illustrated in Figure 5.10, which shows that the try block is executed but no exceptions are thrown, with normal execution continuing after the try-catch construct. This corresponds to Scenario 1 in Figure 5.9.

Example 5.10 try-catch Construct

```
public class Average2 {

    public static void main(String[] args) {
        printAverage(100, 0);                              // (1)
      System.out.println("Exit main().");                   // (2)     }

                          public static void printAverage(int totalSum, int totalNumber) {

        try {                                             // (3)
  int average = computeAverage(totalSum, totalNumber);  // (4)
System.out.println("Average = " +                       // (5)
totalSum + " / " + totalNumber + " = " + average);
} catch (ArithmeticException ae) {                      // (6)
ae.printStackTrace();                              // (7)
    System.out.println("Exception handled in " +
        "printAverage().");            // (8)         }
        System.out.println("Exit printAverage().");           // (9)     }

    public static int computeAverage(int sum, int number) {
   System.out.println("Computing average.");             // (10)
     return sum/number;                                 // (11)     }
}
```

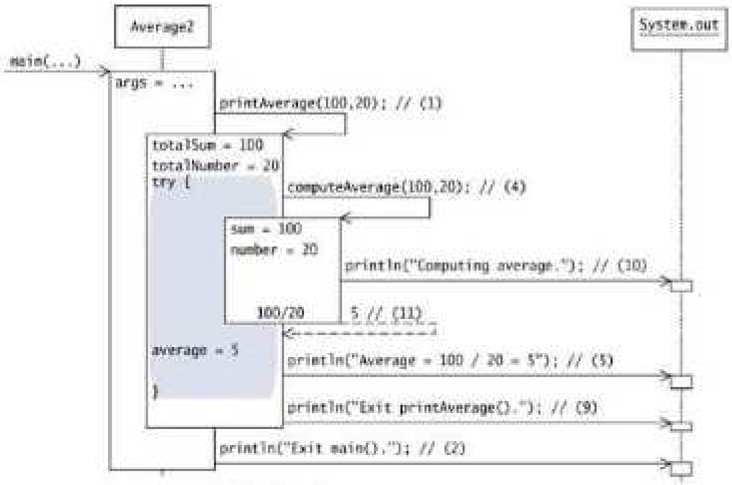
Output from the program, with call printAverage(100, 20) at (1):

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

Output from the program, with call printAverage(100, 0) at (1):

```
Computing average.
java.lang.ArithmeticException: / by zero

at Average2.computeAverage(Average2.java:24)
at Average2.printAverage(Average2.java:11)
at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().

Exit main().
```

Figure 5.10. Exception Handling (Scenario 1)


Average2
System.out
main(...)
args = ...
printAverage(100,20); // (1)
totalSum = 100
totalNumber = 20
try {
computeAverage(100,20); // (4)
sum = 100
number = 20
println("Computing average."); // (10)
100/20
5 // (11)
average = 5
println("Average = 100 / 20 = 5"); // (5)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

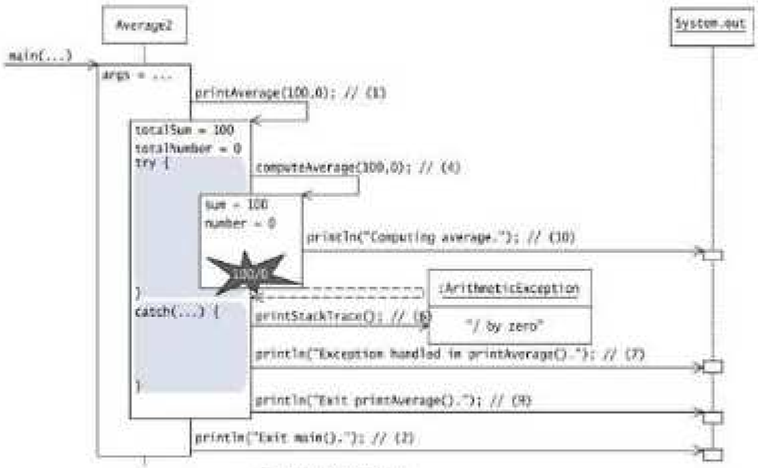
Output from the program:

```
Computing average.
Average = 100 / 20 = 5
Exit printAverage().
Exit main().
```

However, if we run the program in Example 5.10 with the following call in (1):
printAverage(100, 0)

an ArithmeticException is thrown by the integer division in method computeAverage(). From Figure 5.11 we see that the execution of the method computeAverage() is stopped and the exception propagated to method printAverage(), where it is handled by the catch block at (6). Normal execution of the method continues at (9) after the try-catch construct, as witnessed by the output from the statements at (9) and (2). This corresponds to Scenario 2 in Figure 5.9.

Figure 5.11. Exception Handling (Scenario 2)

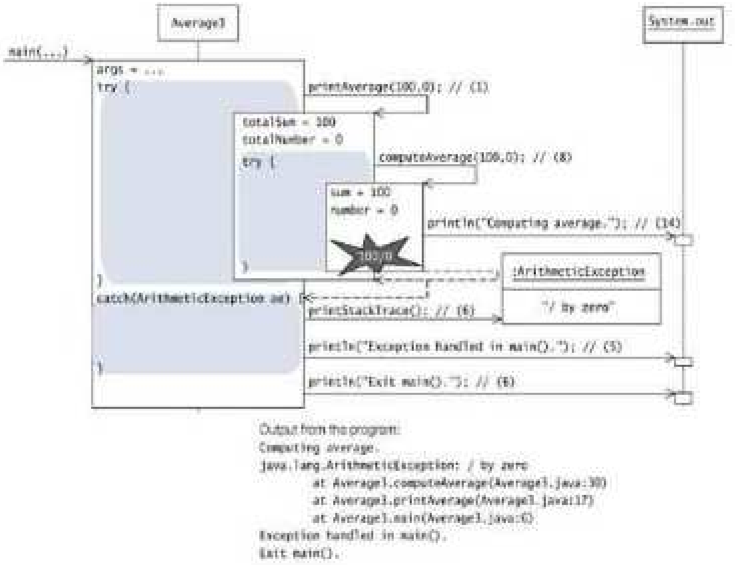

Average2
System.out
main(...)
args = ...
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (4)
sum = 100
number = 0
println("Computing average."); // (10)
100/0
:ArithmeticException
"/ by zero"
}
catch(...) {
printStackTrace(); // (6)
println("Exception handled in printAverage()."); // (7)
}
println("Exit printAverage()."); // (9)
println("Exit main()."); // (2)

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average2.computeAverage(Average2.java:24)
        at Average2.printAverage(Average2.java:11)
        at Average2.main(Average2.java:5)
Exception handled in printAverage().
Exit printAverage().
Exit main().
```

In Example 5.11, the main() method calls the printAverage() method in a try-catch construct at (1). The catch block at (3) is declared to catch exceptions of type ArithmeticException. The printAverage() method calls the computeAverage() method in a try-catch construct at (7), but here the catch block is declared to catch exceptions of type IllegalArgumentException. Execution of the program is illustrated in Figure 5.12, which shows that the ArithmeticException is first propagated to the catch block in the printAverage() method. But since this catch block cannot handle this exception, it is propagated further to the catch block in the main() method, where it is caught and handled. Normal execution continues at (6) after the exception is handled.

Figure 5.12. Exception Handling (Scenario 3)



Average3
System.out
main(...)
args = ...
try {
printAverage(100,0); // (1)
totalSum = 100
totalNumber = 0
try {
computeAverage(100,0); // (8)
sum = 100
number = 0
println("Computing average."); // (14)
100/0
}
}
:ArithmeticException
"/ by zero"
catch(ArithmeticException ae) {
printStackTrace(); // (6)
println("Exception handled in main()."); // (5)
println("Exit main()."); // (6)
}

Output from the program:

```
Computing average.
java.lang.ArithmeticException: / by zero
        at Average3.computeAverage(Average3.java:30)
        at Average3.printAverage(Average3.java:17)
        at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

Note that the execution of try block at (7) in the printAverage() method is never completed: the statment at (9) is never executed. The catch block at (10) is skipped. The execution of the printAverage() method is aborted: the statment at (13) is never executed, and the exception is propagated. This corresponds to Scenario 3 in Figure 5.9.

Example 5.11 Exception Propagation

```
public class Average3 {

   public static void main(String[] args) {
      try {                                          // (1)
printAverage(100, 0);                          // (2)
 } catch (ArithmeticException ae) {                   // (3)
ae.printStackTrace();                          // (4)
 System.out.println("Exception handled in " +
"main().");                    // (5)         }
      System.out.println("Exit main().");                  // (6)     }

   public static void printAverage(int totalSum, int totalNumber) {
 try {                                          // (7)
     int average = computeAverage(totalSum, totalNumber);  // (8)
System.out.println("Average = " +                  // (9)
    totalSum + " / " + totalNumber + " = " + average);
 } catch (IllegalArgumentException iae) {              // (10)
 iae.printStackTrace();                         // (11)
System.out.println("Exception handled in " +
                                "printAverage().");            // (12)        }
        System.out.println("Exit printAverage().");           // (13)     }

    public static int computeAverage(int sum, int number) {
  System.out.println("Computing average.");              // (14)
   return sum/number;                                 // (15)     }
}
```

Output from the program:

```
Computing average.

java.lang.ArithmeticException: / by zero

at Average3.computeAverage(Average3.java:30)
at Average3.printAverage(Average3.java:17)
at Average3.main(Average3.java:6)
Exception handled in main().
Exit main().
```

The scope of the argument name in the catch block is the block itself. As mentioned earlier, the type of the exception object must be assignable to the type of the argument in the catch block. In the body of the catch block, the exception object can be queried like any other object by using the argument name. The javac compiler also complains if a catch block for a superclass exception shadows the catch block for a subclass exception, as the catch block of the subclass exception will never be executed. The following example shows incorrect order of the catch blocks at (1) and (2), which will result in a compile time error: the superclass Exception will shadow the subclass ArithmeticException.

```
...
// Compiler complains
catch (Exception e) {                  // (1) superclass
 System.out.println(e);
} catch (ArithmeticException e) {      // (2) subclass
  System.out.println(e);
}
...
```

finally Block

If the try block is executed, then the finally block is guaranteed to be executed, regardless of whether any catch block was executed. Since the finally block is always executed before control transfers to its final destination, it can be used to specify any clean-up code (for example, to free resources such as, files, net connections).

A try-finally construct can be used to control the interplay between two actions that must be executed in the right order, possibly with other intervening actions.

```
 int sum = -1;
      try {
   sum = sumNumbers();
   // other actions
} finally {
   if (sum >= 0) calculateAverage();
}
```

The code above guarantees that if the try block is entered sumNumbers() will be executed first and then later calculateAverage() will be executed in the finally block, regardless of how execution proceeds in the try block. As the operation in calculateAverage() is dependent on the success of sumNumbers(), this is checked by the value of the sum variable before calling calculateAverage(). catch blocks can, of course, be included to handle any exceptions.

If the finally block neither throws an exception nor executes a control transfer statement like a return or a labeled break, then the execution of the try block or any catch block determines how execution proceeds after the finally block.

· If there is no exception thrown during execution of the try block or the exception has been handled in a catch block, then normal execution continues after the finally block.

• If there is any pending exception that has been thrown and not handled (either due to the fact that no catch block was found or the catch block threw an exception), the method is aborted and the exception is propagated after the execution of the finally block.

If the finally block throws an exception, then this exception is propagated with all its ramifications—regardless of how the try block or any catch block were executed. In particular, the new exception overrules any previously unhandled exception.

If the finally block executes a control transfer statement such as, a return or a labeled break, then this control transfer statement determines how the execution will proceed—regardless of how the try block or any catch block were executed. In particular, a value returned by a return statement in the finally block will supersede any value returned by a return statement in the try block or a catch block.

Output of Example 5.12 shows that the finally block at (9) is executed, regardless of whether an exception is thrown in the try block at (3) or not. If an exception is thrown, it is caught and handled by the catch block at (6). After the execution of the finally block at (9), normal execution continues at (10).

Example 5.12 try-catch-finally Construct

```
public class Average4 {

    public static void main(String[] args) {
        printAverage(100, 20);                              // (1)
   System.out.println("Exit main().");                   // (2)     }

    public static void printAverage(int totalSum, int totalNumber) {
   try {                                                   // (3)


            int average = computeAverage(totalSum, totalNumber);  // (4)
System.out.println("Average = " +                       // (5)
totalSum + " / " + totalNumber + " = " + average);         }
catch (ArithmeticException ae) {                     // (6)
 ae.printStackTrace();                                 // (7)
 System.out.println("Exception handled in " +
      "printAverage().");            // (8)         }
finally {                                              // (9)
      System.out.println("Finally done.");         }
        System.out.println("Exit printAverage().");           // (10)     }

    public static int computeAverage(int sum, int number) {
   System.out.println("Computing average.");             // (11)
 return sum/number;                                    // (12)     }
}
```

Output from the program, with call printAverage(100, 20) at (1):

```
Computing average.

Average = 100 / 20 = 5
Finally done.
Exit printAverage().
Exit main().
```

Output from the program, with call printAverage(100, 0) at (1):

```
Computing average.
java.lang.ArithmeticException: / by zero
at Average4.computeAverage(Average4.java:26)
at Average4.printAverage(Average4.java:11)
  at Average4.main(Average4.java:5)
Exception handled in printAverage().
```

```
Finally done.
Exit printAverage().
Exit main().
```

On exiting from the finally block, if there is any pending exception, the method is aborted and the exception propagated as explained earlier. This is illustrated in Example 5.13. The method printAverage() is aborted after the finally block at (6) has been executed, as the ArithmeticException thrown at (9) is not handled by any method. In this case, the exception is handled by the default exception handler. Notice the difference in the output from Example 5.12 and Example 5.13.

Example 5.13 try-finally Construct

```
public class Average5 {

    public static void main(String[] args) {
        printAverage(100, 0);                              // (1)
  System.out.println("Exit main().");                      // (2)     }

    public static void printAverage(int totalSum, int totalNumber) {
  try {                                               // (3)
  int average = computeAverage(totalSum, totalNumber);  // (4)
System.out.println("Average = " +                         // (5)
   totalSum + " / " + totalNumber + " = " + average);          }
finally {                                             // (6)
 System.out.println("Finally done.");          }
        System.out.println("Exit printAverage().");           // (7)     }

    public static int computeAverage(int sum, int number) {
  System.out.println("Computing average.");               // (8)
  return sum/number;                                   // (9)     }
}
```

```
Output from the program:
Computing average.
Finally done.
Exception in thread "main" java.lang.ArithmeticException: / by zero
  at Average5.computeAverage(Average5.java:21)
  at Average5.printAverage(Average5.java:10)
at Average5.main(Average5.java:4)
```

Example 5.14 shows how the execution of a control transfer statement such as a return in the finally block affects the program execution. The first output from the program shows that the average is computed, but the value returned is from the return statement at (8) in the finally block, not from the return statement at (6) in the try block. The second output shows that the ArithmeticException thrown in the computeAverage() method and propagated to the printAverage() method, is nullified by the return statement in the finally block. Normal execution continues after the return statement at (8), with the value 0 being returned from the printAverage() method.

Example 5.14 finally Block and the return Statement

```
    public class Average6 {

       public static void main(String[] args) {

     System.out.println("Average: " + printAverage(100, 20));  // (1)
System.out.println("Exit main().");                      // (2)     }

 public static int printAverage(int totalSum, int totalNumber) {
   int average = 0;
     try {                                         // (3)
average = computeAverage(totalSum, totalNumber);      // (4)
 System.out.println("Average = " +                   // (5)
    totalSum + " / " + totalNumber + " = " + average);
 return average;                               // (6)
 } finally {                                   // (7)
 System.out.println("Finally done.");
return average*2;                              // (8)        }
   }

   public static int computeAverage(int sum, int number) {

    System.out.println("Computing average.");                // (9)

           return sum/number;                           // (10)     }

}
```

Output from the program, with call printAverage(100, 20) in (1):

```
Computing average.
Average = 100 / 20 = 5
Finally done.
Average: 10
Exit main().
```

Output from the program, with call printAverage(100, 0) in (1):

```
Computing average.
Finally done.
Average: 0
Exit main().
```

5.8 throw Statement

Earlier examples in this chapter have shown how an exception is thrown implicitly during execution. A program can explicitly throw an exception using the throw statement. The general format of the throw statement is as follows:

```
throw <object reference expression>;
```

The compiler ensures that the <object reference expression> is of type Throwable class or one of its subclasses. At runtime a NullPointerException is thrown if the <object reference expression> is null. This ensure that a Throwable will always be propagated. A detail message is often passed to the constructor when the exception object is created.

```
throw new ArithmeticException("Integer division by 0");
```

When an exception is thrown, normal execution is suspended. The runtime system proceeds to find a catch block that can handle the exception. The search starts in the context of the current try block, propagating to any enclosing try blocks and through the runtime stack to find a handler for the exception. Any associated finally block of a try block encountered along the search path is executed. If no handler is found, then the exception is dealt with by the default exception handler at the top level. If a handler is found, execution resumes with the code in its catch block.

In Example 5.15, an exception is thrown using a throw statement at (17). This exception is propagated to the main() method where it is caught and handled by the catch block at (3). Note that the finally blocks at (6) and (14) are executed. Execution continues normally from (7).

Example 5.15 Throwing Exceptions

```
public class Average7 {
   public static void main(String[] args) {
       try {                                          // (1)
printAverage(100, 0);                                 // (2)
} catch (ArithmeticException ae) {                    // (3)
 ae.printStackTrace();                                // (4)
 System.out.println("Exception handled in " +         // (5)
   "main().");
       } finally {
           System.out.println("Finally in main().");  // (6)
 }
       System.out.println("Exit main().");            // (7)
  }
   public static void printAverage(int totalSum, int totalNumber) {
 try {                                                // (8)
 int average = computeAverage(totalSum, totalNumber);   // (9)
System.out.println("Average = " +                     // (10)
  totalSum + " / " + totalNumber + " = " + average);
} catch (IllegalArgumentException iae) {                    // (11)
 iae.printStackTrace();                                // (12)
  System.out.println("Exception handled in " +          // (13)
"printAverage().");
        } finally {

            System.out.println("Finally in printAverage().");      // (14)         }
        System.out.println("Exit printAverage().");               // (15)     }
    public static int computeAverage(int sum, int number) {
System.out.println("Computing average.");
if (number == 0)                                        // (16)
 throw new ArithmeticException("Integer division by 0");// (17)
   return sum/number;                                      // (18)     }
}
```

Output from the program:

```
Computing average.
Finally in printAverage().
java.lang.ArithmeticException: Integer division by 0
at Average7.computeAverage(Average7.java:35)
 at Average7.printAverage(Average7.java:19)
at Average7.main(Average7.java:6)
Exception handled in main().
Finally in main().

Exit main().
```

5.9 throws Clause

A throws clause can be specified in the method protoype.

```
                     ... someMethod(...)
                    throws <ExceptionType1>, <ExceptionType2>,..., <ExceptionTypen> { ... }
```

Each <ExceptionTypei> declares a checked exception. The compiler enforces that the checked exceptions thrown by a method are limited to those specified in its throws clause. Of course, the method can throw exceptions that are subclasses of the checked exceptions in the throws clause. This is permissable since exceptions are objects, and a subclass object can polymorphically act as an object of its superclass. The throws clause can have unchecked exceptions specified, but this is seldom used and the compiler does not check them.

Any method that can cause a checked exception to be thrown, either directly by using the throw statement or indirectly by invoking other methods that can throw such an exception, must deal with the exception in one of three ways. It can

· use a try block and catch the exception in a handler and deal with it
· use a try block and catch the exception in a handler, but throw another exception that is either unchecked or declared in its throws clause
· explicitly allow propagation of the exception to its caller by declaring it in the throws clause of its method protoype

This mechanism ensures that, regardless of the path of execution, a checked exception will be dealt with in some way. It aids development of robust programs, as allowance can be made for many contingencies. Native methods can also declare checked exceptions in their throws clause, but the compiler is not able to check them for consistency.

A new checked exception is defined in Example 5.16. The checked exception class IntegerDivisionByZero is defined at (11) by extending the Exception class.

In Example 5.16, the method main() calls the method printAverage() in a try block at (1). In the if statement at (9), the method computeAverage() throws the checked exception IntegerDivisionByZero defined at (11). Neither the computeAverage() method nor the printAverage() method catch the exception, but instead throw it to their caller, as declared in the throws clause in their headers at (6) and (8). The exception propagates to the main() method. Since the printAverage() method was called from the context of the try block at (1) in the main() method, the exception is successfully matched with its catch block at (3). The exception is handled and the finally block at (4) executed, with normal execution proceeding at (5). If the method main() did not catch the exception, it would have to declare this exception in a throws clause. In that case, the exception would end up being taken care of by the default exception handler.

Example 5.16 throws Clause

```
public class Average8 {
   public static void main(String[] args) {
      try {                                    // (1)
 printAverage(100, 0);                          // (2)
} catch (IntegerDivisionByZero idbze) {                 // (3)
idbze.printStackTrace();
         System.out.println("Exception handled in " +
 "main().");
      } finally {                               // (4)

 System.out.println("Finally done in main().");        }
```

```
        System.out.println("Exit main().");                    // (5)
}

    public static void printAverage(int totalSum, int totalNumber)
  throws IntegerDivisionByZero {                      // (6)
        int average = computeAverage(totalSum, totalNumber);
System.out.println("Average = " +
            totalSum + " / " + totalNumber + " = " + average);
  System.out.println("Exit printAverage().");             // (7)     }

    public static int computeAverage(int sum, int number)
  throws IntegerDivisionByZero {                      // (8)
        System.out.println("Computing average.");
 if (number == 0)                                    // (9)
    throw new IntegerDivisionByZero("Integer Division By Zero");
 return sum/number;                                  // (10)
}
}

 class   IntegerDivisionByZero   extends   Exception   {                                      //   (11)
IntegerDivisionByZero(String str) { super(str); }           // (12) }
```

Output from the program:

```
Computing average.
IntegerDivisionByZero: Integer Division By Zero
 at Average8.computeAverage(Average8.java:33)
 at Average8.printAverage(Average8.java:22)
 at Average8.main(Average8.java:7)
Exception handled in main().
Finally done in main().
Exit main().
```

The exception type specified in the throws clause in the method protoype can be a superclass type of the actual exceptions thrown, that is, the exceptions thrown must be assignable to the type of the exceptions specified in the throws clause. If a method can throw exceptions of the type A, B, and C where these are subclasses of type D, then the throws clause can either specify A, B, and C or just specify D. In the printAverage() method, the method protoype could specify the superclass Exception of the subclass IntegerDivisionByZero in a throws clause.

```
public static void printAverage(int totalSum, int totalNumber)
throws Exception { /* ... */ }
```

It is generally a bad programming style to specify exception superclasses in the throws clause of the method protoype, when the actual exceptions thrown in the method are instances of their subclasses. Programmers will be deprived of information about which specific subclass exceptions can be thrown, unless they have access to the source code.

A subclass can override a method defined in its superclass by providing a new implementation. What happens when an inherited method with a list of exceptions in its throws clause is overridden in a subclass? The method definition in the subclass can only specify a subset of the checked exception classes (including their subclasses) from the throws clause of the inherited method in the superclass. This means that an overriding method cannot allow more checked exceptions in its throws clause than the inherited method does. Allowing more checked exceptions in the overriding method would create problems for clients who already deal with the exceptions specified in the inherited method. Such clients would be ill prepared if an object of the subclass (under the guise of polymorphism) threw a checked exception they were not prepared for.

```
class A {
   // ...
   protected void superclassMethodX()
      throws FirstException, SecondException, ThirdException {/* ... */} // (1)     // ...
}
class B extends A {
   // ...
   protected void superclassMethodX()
      throws FirstException, ThirdException { /* ... */ }              // (2)     // ...
}
```

In the previous code, the method superclassMethodX in superclass A is overridden in subclass B. The throws clause of the method in subclass B at (2) is a subset of the exceptions specified for the method in the superclass at (1).

5.10 Assertions

Assertions in Java can be used to document and validate assumptions made about the state of the program at designated locations in the code. Each assertion contains a boolean expression that is expected to be true when the assertion is executed. If this assumption is false, the system throws a special assertion error. The assertion facility uses the exception handling mechanism to propagate the error. The assertion facility can be enabled or disabled at runtime.

The assertion facility is an invaluable aid in implementing correct programs (i.e., programs that adhere to their specification). It should not be confused with the exception handling mechanism that aids in developing robust programs (i.e., programs that handle unexpected conditions gracefully). Used judiciously, the two mechanisms facilitate programs that are reliable.

assert Statement and AssertionError Class

The following two forms of the assert statement can be used to specify assertions:

assert <boolean expression> ;   // the simple form

assert <boolean expression> : <message expression> ;   // the augmented form

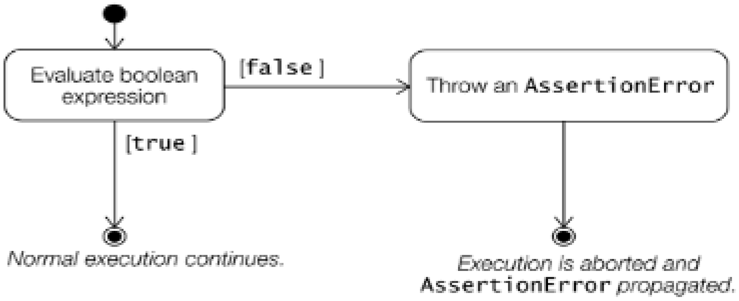
If assertions are enabled (see p. 212), the execution of an assert statement proceeds as shown in Figure 5.13. The two forms are essentially equivalent to the following code, respectively:

if (<assertions enabled> && !<boolean expression>)   // the simple form   throw new AssertionError();

if (<assertions enabled> && !<boolean expression>)   // the augmented form

throw new AssertionError(<message expression>);

Figure 5.13. Execution of the Simple assert Statement (When Assertions Are Enabled)


Evaluate boolean expression
[false]
Throw an AssertionError
[true]
Normal execution continues.
Execution is aborted and AssertionError propagated.

If assertions are enabled, then <boolean expression> is evaluated. If its value is true, execution continues normally after the assert statement. However, if it is false, an AssertionError is thrown and propagated. In the simple form, the AssertionError does not provide any detailed message about the assertion failure.

The augmented form provides a <message expression> that can be used to provide a detailed error message. In the augmented form, if the assertion is false, the <message expression> is evaluated and its value passed to the appropriate AssertionError constructor. The <message expression> must evaluate to a value (i.e., either a primitive or a reference value). The AssertionError constructor invoked converts the value to a textual representation. In particular, the <message expression> cannot call a method that is declared void. The compiler will flag this as an error.

Lines (2), (3), and (4) in class Speed are assertion statements. In this particular context of calculating the speed, it is required that the values fulfill the criteria in lines (2), (3) and (4) in the private method calcSpeed(). Lines (2) and (4) use the simple form.

```
        assert distance >= 0.0;                    // (2) ...

        assert speed >= 0.0;                       // (4)
```

Line (3) uses the augmented form.

```
assert time > 0.0 : "Time is not a positive value: " + time;  // (3)
```

Line (3) is equivalent to the following line of code, assuming assertions have been enabled at runtime:

```
if (time <= 0.0) throw new AssertionError("Time is not a positive value: " + time);
```

The java.lang.AssertionError class is a subclass of java.lang.Error. This makes AssertionError exceptions unchecked. They could be explicitly caught and handled using the try-catch construct. The execution would then continue normally, as one would expect. However, since Error exceptions are seldom caught and handled by the program, the same applies to AssertionError exceptions. Catching these exceptions would defeat the whole purpose of the assertion facility.

In addition to the default constructor (invoked by the simple assert form), the AssertionError class provides seven single-parameter constructors: six for the primitive data types (byte and short being promoted to int) and one for object references. The type of the <message expression> used in the augmented assertion statement determines which of the overloaded constructors is invoked. It is not possible to query the AssertionError object for the actual value passed to the constructor. However, the method getMessage() will return the textual representation of the value.

Example 5.17 Assertions

```
public class Speed {
 public static void main(String[] args) {
       Speed objRef = new Speed();
     double speed = objRef.calcSpeed(-12.0, 3.0);              // (1a)
 // double speed = objRef.calcSpeed(12.0, -3.0);             // (1b)
    // double speed = objRef.calcSpeed(12.0, 2.0);             // (1c)
   // double speed = objRef.calcSpeed(12.0, 0.0);             // (1d)
System.out.println("Speed (km/h): " + speed);     }

    /** Requires distance >= 0.0 and time > 0.0 */
private double calcSpeed(double distance, double time) {
 assert distance >= 0.0;                              // (2)
 assert time >0.0 : "Time is not a positive value: " + time;  // (3)
double speed = distance / time;
       assert speed >= 0.0;                              // (4)
   return speed;
    }
}
```

Compiling Assertions

The assertion facility was introduced in J2SE 1.4. At the same time, two new options for the javac compiler were introduced for dealing with assertions in the source code.

Option -source 1.4

The javac compiler distributed with the Java SDK v1.4 will only compile assertions if the option -source 1.4 is used on the command-line:

```
>javac -source 1.4 Speed.java
```

This also means that incorrect use of the keyword assert will be flagged as an error, for example, if assert is used as an identifier. The following program

```
   public class Legacy {
   public static void main(String[] args) {
      int assert = 2003;
      System.out.println("The year is: " + assert);     }
}
```

when compiled, results in two errors:

```
>javac -source 1.4 Legacy.java
Legacy.java:4: as of release 1.4, assert is a keyword, and may not be used as an  identifier
      int assert = 2003;
         ^
Legacy.java:5: as of release 1.4, assert is a keyword, and may not be used as an
 identifier
      System.out.println("The year is: " + assert);                              ^
2 errors
```

Option -source 1.3

The default behavior of the javac compiler is equivalent to using the option -source 1.3 on the command-line.

```
>javac -speed 1.3 Speed.java
Speed.java:14: warning: as of release 1.4, assert is a keyword, and may not be used as an
 identifier
 assert distance >= 0.0;                              // (2)
  ^
Speed.java:15: ';' expected
      assert distance >= 0.0;                              // (2)
  ^

...
9 errors
3 warnings
```

The compiler will reject assert statements. It will also warn about the use of the keyword assert as an identifier. In other words, source code that contains the keyword assert as an identifier will compile (barring any other errors), but it will also result in a warning. Compiling and running the Legacy class above gives the following results:

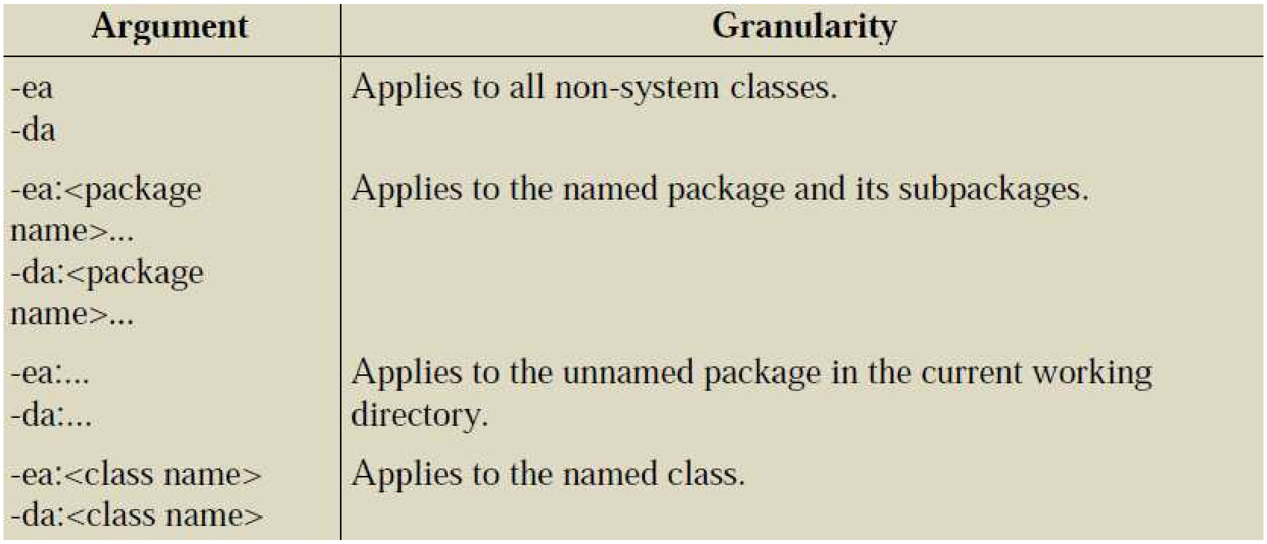
```
 >javac -source 1.3 Legacy.java
Legacy.java:4: as of release 1.4, assert is a keyword, and may not be used as an
identifier
      int assert = 2003;
         ^
Legacy.java:5: as of release 1.4, assert is a keyword, and may not be used as an
 identifier
      System.out.println("The year is: " + assert);                              ^

2 warnings
>java Legacy
The year is: 2003
```

Runtime Enabling and Disabling of Assertions

Enabling assertions means they will be executed at runtime. By default, assertions are disabled. Their execution is then effectively equivalent to empty statements. This means that disabled assertions carry an insignificant performance penalty, although they add storage overhead to the byte code of a class. Typically, assertions are enabled during development and left disabled once the program is deployed.

Two command switches are provided by the java tool to enable and disable assertions with various granularities. The switch -enableassertions, or its short form -ea, enables assertions, and the switch -disableassertions, or its short form -da, disables assertions at various granularities. The granularities that can be specified are shown in Table 5.2.

Table 5.2. Granularities for Enabling and Disabling Assertions at Runtime



Argument	Granularity
-ea -da	Applies to all non-system classes.
-ea:<package name>... -da:<package name>...	Applies to the named package and its subpackages.
-ea:... -da:...	Applies to the unnamed package in the current working directory.
-ea:<class name> -da:<class name>	Applies to the named class.

Assertion Execution for All Non-system Classes

The -ea option means that all non-system classes loaded during the execution of the program have their assertions enabled. A system class is a class that is in the Java platform libraries. For example, classes in the java.* packages are system classes. A system class is loaded directly by the JVM.

Note that class files not compiled with a J2SE 1.4-compatible compiler are not affected, whether assertions are enabled or disabled. Also, once a class has been loaded and initialized at runtime, its assertion status cannot be changed.

Assuming that the file Speed.java has been compiled with the -source 1.4 option, all assertions in non-system classes required for execution (of which Speed class is one) can be enabled, and the program run as follows:

```
     >java -ea Speed
    java.lang.AssertionError
    at Speed.calcSpeed(Speed.java:14)
    at Speed.main(Speed.java:6)
Exception in thread "main"
```

Since the distance is negative in line (1a), the assertion in line (2) fails in Example 5.17. An AssertionError is thrown, which is propagated, being finally caught by the default exception handler and resulting in the stack trace being printed on the terminal.

All assertions (in all non-system classes) can be disabled during the execution of the Speed class.

```
>java -da Speed
Speed (km/h): -4.0
```

In this case, this is effectively equivalent to running the program with neither the -ea nor the -da options.

```
>java Speed
Speed (km/h): -4.0
```

If we comment-out line (1a) and uncomment line (1b) in Example 5.17 and run the program with the options enabled, we get the following behavior from the program.

```
>java -ea Speed
java.lang.AssertionError: Time is not a positive value: -3.0
   at Speed.calcSpeed(Speed.java:15)
      at Speed.main(Speed.java:7)
Exception in thread "main"
```

We see that the value of the <message expression> in the augmented assertion in line (3) is written on the terminal, together with the stack trace, because this assertion failed. The augmented form is recommended, as it allows a detailed error message to be included in reporting the assertion failure.

Assertion Execution at the Package Level

Assume that we have a program called Trickster in the unnamed package, that uses the package hierarchy for the wizard package shown in Figure 4.3.

The following command-line will only enable assertions for all classes in the package wizard.pandorasBox and its subpackage wizard.pandorasBox.artifacts. The assertions in the class Trickster are not enabled.

```
>java -ea:wizard.pandorasBox... Trickster
```

Without the ... notation, the package name will be interpreted as a class name. Non-existent package names specified in the command line are silently accepted, but simply have no consequences under execution.

The following command-line will only enable assertions in the unnamed package, and, thereby, the assertions in the class Trickster since this class resides in the unnamed package.

>java -ea:... Trickster

Note that the package switch applies to the package specified and all its subpackages, recursively.

Assertion Execution at the Class Level

The following command line will only enable assertions in the Trickster class.

```
>java -ea:Trickster Trickster
```

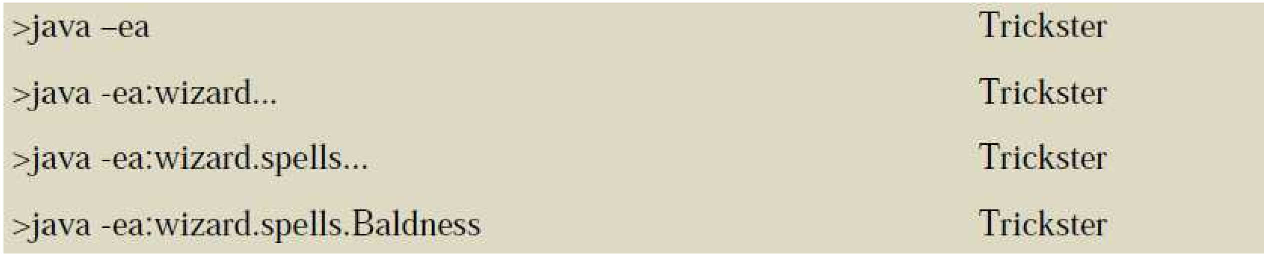
The following command line will only enable assertions in the named class wizard.pandorasBox.artifacts.Ailment and no other class.

```
>java -ea:wizard.pandorasBox.artifacts.Ailment Trickster
```

A java command can contain multiple instances of the switches, each specifying its own granularity. The switches are then processed in order of their specification from left to right, before any classes are loaded. The latter switches take priority over former switches. This allows a fine-grained control of what assertions are enabled at runtime. The following command line will enable assertions for all classes in the package wizard.pandorasBox and its subpackage wizard.pandorasBox.artifacts, but disable them in the class wizard.pandorasBox.artifacts.Ailment.

```
>java -ea:wizard.pandorasBox... -da:wizard.pandorasBox.artifacts.Ailment Trickster
```

The following switches all enable assertions in the class wizard.spells.Baldness.



```
>java –ea                                   Trickster
>java -ea:wizard...                         Trickster
>java -ea:wizard.spells...                  Trickster
>java -ea:wizard.spells.Baldness            Trickster
```

It is worth noting that inheritance has no affect on the execution of assertions. Assertions are enabled or disabled on per-class basis. Whether assertions in the superclass will be executed through code inherited by the subclass, depends entirely on the superclass. In the following command line, assertions from the superclass wizard.pandorasBox.artifacts.Ailment will not be executed, although assertions for the subclass wizard.spells.Baldness are enabled:

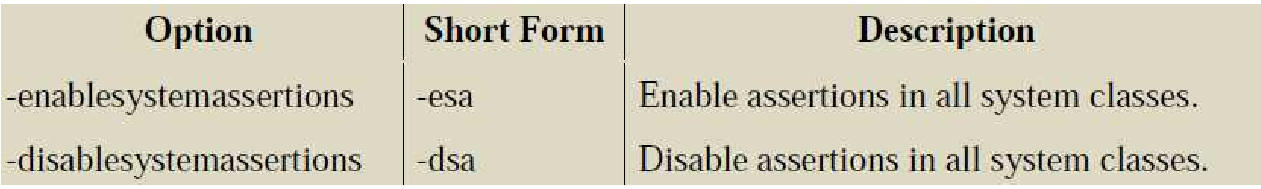
```
>java -ea -da:wizard.pandorasBox.artifacts.Ailment Trickster
```

Assertion Execution for All System Classes

In order to enable or disable assertions in all system classes, we can use the switches shown in Table 5.3. Enabling assertions in system classes can be useful to shed light on internal errors reported by the JVM. In the following command line, the first switch will enable assertions for all system classes. The second switch will enable assertions in the package wizard and its subpackages wizard.pandorasBox, wizard.pandorasBox.artifacts and wizard.spells, but the third switch will disable them in the package wizard.pandorasBox.artifacts.

```
>java -esa -ea:wizard... -da:wizard.pandorasBox.artifacts... Trickster
```

Table 5.3. Enabling and Disabling Assertions in All System Classes at Runtime



Option	Short Form	Description
-enablesystemassertions	-esa	Enable assertions in all system classes.
-disablesystemassertions	-dsa	Disable assertions in all system classes.

Using Assertions

Assertions should have no side effects that can produce adverse behavior in the code, whether enabled or not. The assertion facility is a defensive mechanism, meaning that it should only be used to test the code, and should not be employed after the code is delivered. The program should exhibit the same behavior whether assertions are enabled or disabled. The program should not rely on any computations done within an assertion statement. With assertions enabled, the following statement would be executed, but if assertions were disabled, it could have dire consequences.

```
assert reactor.controlCoreTemperature();
```

Assertions should also not be used to validate information supplied by a client. A typical example is argument checking in public methods. Argument checking is part of such a method's contract, which could be violated if the assertions were disabled. Another drawback is that assertion failures can only provide limited information, in the form of an AssertionError, about the cause of any failure. Appropriate argument checking can provide more suitable information about erroneous arguments, in the form of specific exceptions such as IllegalArgumentException, IndexOutOfBoundsException, or NullPointerException.

The rest of this section illustrates useful idioms that employ assertions.

Internal Invariants

Very often assumptions about the program are documented as comments in the code. The following code makes the assumption in line (1) that variable status must be negative for the else clause to be executed.

```
int status = ref1.compareTo(ref2);
if (status == 0) {
   ...
} else if (status > 0) {
   ...
} else { // (1) status must be negative.
    ...
}
```

This assumption is an internal invariant and can be verified using an assertion, as shown in line (2) below.

```
int status = ref1.compareTo(ref2);
if (status == 0) {
    ...
} else if (status > 0) {
    ...
} else {
    assert status < 0 : status; // (2)
    ...
}
```

Often an alternative action is chosen, based on a value that is guaranteed to be one of a small set of predefined values. A switch statement with no default clause is a typical example. The value of the switch expression is guaranteed to be one of the case labels and the default case is omitted, as the following code shows.

```
 switch (trinityMember) {
    case Housefather:
        ...
        break;
    case THE_SON:
        ...
        break;
    case THE_HOLY_GHOST:
        ...
        break;
}
```

A default clause that executes an assertion can be used to formulate this invariant.

```
default:
 assert false : trinityMember;
```

If assertions are enabled, an AssertionError will signal the failure in case the trinity no longer holds.

However, the previous code causes a compile-time error in a non-void method if all case labels return a value and no return statement follows the switch statement.

```
switch (trinityMember) {
    case THE_FATHER:
        return psalm101;
```

```
   case THE_SON:
      return psalm102;
   case THE_HOLY_GHOST:
      return psalm103;
   default:
      assert false: trinityMember;
}
return psalm100;        // (3) Compile time error if commented out.
```

Without line (3) and with assertions disabled, the method could return without a value, violating the fact that it is a non-void method. Explicitly throwing an AssertionError rather than using an assert statement in the default clause, would be a better option in this case.

```
               default:
                                   throw new AssertionError(trinityMember);
```

Control Flow Invariants

Control flow invariants can be used to test assumptions about the flow of control in the program. The following idiom can be employed to explicitly test that certain locations in the code will never be reached.

```
assert false : "This line should never be reached.";
```

If program control does reach this statement, assertion failure will detect it.

In the following code, the assumption is that execution never reaches the end of the method declaration indicated by line (1).

```
private void securityMonitor() {
   // ...
   while (alwaysOnDuty) {
      // ...
      if (needMaintenance)
         return;
      // ...
   }
   // (1) This line should never be reached.
}
```

The previous assertion can be inserted after the comment at line (1) to check the assumption.

Care should be taken in using this idiom, as the compiler can flag the assert statement at this location as being unreachable. For example, if the compiler can deduce that the while condition will always be true, it will flag the assert statement as being unreachable.

Preconditions and Postconditions

The assertion facility can be used to practice a limited form of programming-by-contract. For example, the assertion facility can be used to check that methods comply with their contract.

Preconditions define assumptions for the proper execution of a method when it is invoked. As discussed earlier, assertions should not be used to check arguments in public methods. For non-public methods, preconditions can be checked at the start of method execution.

```
private void adjustReactorThroughput(int increment) {
 // Precondition:
   assert isValid(increment) : "Throughput increment invalid.";
 // Proceed with the adjustment.
   // ...
}
```

Postconditions define assumptions about the successful completion of a method. Postconditions in any method can be checked by assertions executed just before returning from the method. For example, if the method adjustReactorThroughPut() guarantees that the reactor core is in a stable state after its completion, we can check this postcondition using an assertion.

```
private void adjustReactorThroughput(int increment) {
 // Precondition:
                          assert isValid(increment) : "Throughput increment invalid.";

   // Proceed with the adjustment.
   // ...
   // Postcondition -- the last action performed before returning.
  assert isCoreStable() : "Reactor core not stable."; }
```

Other Uses

If minimizing the size of the class file is crucial, then the following conditional compilation idiom should be used to insert assertions in the source code:

```
final static boolean COMPILE_ASSERTS = false; ...
if (COMPILE_ASSERTS)
   assert whatEverYouWant;
```

```
// Not compiled if COMPILE_ASSERTS is false.
...
```

It is possible to enforce that a class be loaded and initialized only if its assertions are enabled. The idiom for this purpose uses a static initializer.

```
static {  // Static initializer
    boolean assertsAreEnabled = false;  // (1)
 assert assertsAreEnabled = true;    // (2) utilizing side effect
 if (!assertsAreEnabled)              // (3)
        throw new AssertionError("Enable assertions!"); }
```

Line (1) sets the local variable assertsAreEnabled to false. If assertions are enabled, line (2) is executed. The assignment operator sets the variable assertsAreEnabled to true as a side effect of evaluating the boolean expression that has the value true. The assertion in line (2) is, of course, true. No exception is thrown by the if statement in line (3). However, if assertions are disabled, line (2) is never executed. As the variable assertsAreEnabled is false, the if statement in line (3) throws an exception. The static initializer is placed first in the class declaration, so that it is executed first during class initialization.

06

Object-oriented Programming

6.1 Single Implementation Inheritance

One of the fundamental mechanisms for code reuse in OOP, is inheritance. It allows new

classes to be derived from an existing class. The new class (a.k.a   subclass, subtype, derived class, child class) can inherit members from the old class (a.k.a. superclass, supertype, base class, parent class). The subclass can add new behavior and properties, and under certain circumstances, modify its inherited behavior.

In Java, implementation inheritance is achieved by extending classes (i.e., adding new fields and methods) and modifying inherited members. Inheritance of members is closely tied to their declared accessibility. If a superclass member is accessible by its simple name in the subclass (without the use of any extra syntax like super), then that member is considered inherited. This means that private, overridden, and hidden members of the superclass are not inherited. Inheritance should not be confused with the existence of such members in the state of a subclass object.

The superclass is specified using the extends clause in the header of the subclass declaration. The subclass only specifies the additional new and modified members in its class body. The rest of its declaration is made up of its inherited members. If no extends clause is specified in the header of a class declaration, then the class implicitly inherits from the java.lang.Object class. This implicit inheritance is assumed in the declaration of the Light class at (1) in Example 6.1. Also in Example 6.1, the subclass TubeLight at (2) explicitly uses the extends clause and only specifies additional members to what it already inherits from the superclass Light (which, in turn, inherits from the Object class). Members of the superclass Light that are accessible by their simple names in the subclass TubeLight, are inherited by the subclass.

Private members of the superclass are not inherited by the subclass and can only be indirectly accessed. The private field indicator of the superclass Light is not inherited, but exists in the subclass object and is indirectly accessible.

Using appropriate accessibility modifiers, the superclass can limit which members can be accessed directly and, thereby, inherited by its subclasses. As shown in Example 6.1, the subclass can use the inherited members as if they were declared in its own class body. This is not the case for members that are declared private in the superclass. Members that have package accessibility in the superclass are also not inherited by subclasses in other packages, as these members are only accessible by their simple names in subclasses within the same package as the superclass. Since constructors and initializer blocks are not members of a class, they are not inherited by a subclass.

Example 6.1 Extending Classes: Inheritance and Accessibility

```
            class Light {                    // (1)
          // Instance fields
           int     noOfWatts;    // wattage
   private   boolean indicator;    // on or off
protected String  location;     // placement
   // Static fields
   private static int counter;     // no. of Light objects created
   // Constructor
   Light() {
      noOfWatts = 50;
      indicator = true;
      location  = "X";
      counter++;
   }

   // Instance methods

   public  void    switchOn()  { indicator = true; }
 public  void    switchOff() { indicator = false; }
public  boolean isOn()      { return indicator; }
  private void    printLocation() {
       System.out.println("Location: " + location);     }

   // Static methods
   public static void writeCount() {
       System.out.println("Number of lights: " + counter);     }
   //...
}

class TubeLight extends Light {     // (2)
 Subclass uses the extends clause.     // Instance fields
   private int tubeLength = 54;
   private int colorNo    = 10;


   // Instance methods

   public int getTubeLength() { return tubeLength; }
```

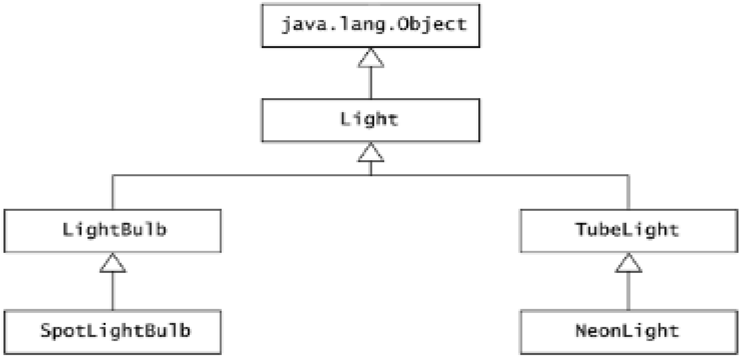
```
    public void printInfo() {
        System.out.println("Tube length: "  + getTubeLength());
 System.out.println("Color number: " + colorNo);
System.out.println("Wattage: "      + noOfWatts); // Inherited.
 //  System.out.println("Indicator: "    + indicator); // Not Inherited.
System.out.println("Indicator: "    + isOn());    // Inherited.
  System.out.println("Location: "     + location);  // Inherited.
//  printLocation();                              // Not Inherited.
 //  System.out.println("Counter: "    + counter);     // Not Inherited.

  writeCount();                                 // Inherited.

    }
    // ...
}

public class Utility {               // (3)
    public static void main(String[] args) {
       new TubeLight().printInfo();
    }
}
```

Output from the program:

```
Tube length: 54
Color number: 10
Wattage: 50
Indicator: true
Location: X
Number of lights: 1
```

A class in Java can only extend one other class; that is, it can only have one immediate superclass. This kind of inheritance is sometimes called single or linear implementation inheritance. The name is appropriate, as the subclass inherits the implementations of its superclass members. The inheritance relationship can be depicted as an inheritance hierarchy (also called class hierarchy). Classes higher up in the hierarchy are more generalized, as they abstract the class behavior. Classes lower down in the hierarchy are more specialized, as they customize the inherited behavior by additional properties and behavior. Figure 6.1 illustrates the inheritance relationship between the class Light, which represents the more general abstraction, and its more specialized subclasses. The java.lang.Object class is always at the top of any Java inheritance hierarchy, as all classes, with the exception of the Object class itself, inherit (either directly or indirectly) from this

Figure 6.1. Inheritance Hierarchy



Inheritance defines the relationship is-a (also called the superclass–subclass relationship) between a superclass and its subclasses. This means that an object of a subclass can be used wherever an object of the superclass can be used. This is often employed as a litmus test for using inheritance. It has particular consequences on how objects can be used. An object of the TubeLight class can be used wherever an object of the superclass Light can be used. An object of the TubeLight class is-an object of the superclass Light. The inheritance relationship is transitive: if class B extends class A, then a class C, which extends class B, will also inherit from class A via class B. An object of the SpotLightBulb class is-a object of the class Light. The is-a relationship does not hold between peer classes: an object of the LightBulb class is not an object of the class TubeLight and vice versa.

Whereas inheritance defines the relationship is-a between a superclass and its subclasses, aggregation defines the relationship has-a (a.k.a. whole–part relationship) between an instance of a class and its constituents (a.k.a. parts). An instance of class Light has the following parts: a field to store its wattage (noOfWatts), a field to store whether it is on or off (indicator), and a String object to store its location (denoted by the field reference location). In Java, a composite object cannot contain other objects. It can only have references to its constituent objects. This relationship defines an aggregation hierarchy that embodies the has-a relationship. Constituent objects can be shared between objects and their lifetimes can be independent of the lifetime of the composite object.

Object-oriented Programming Concepts

The example in this section illustrates basic OOP concepts, and subsequent sections in this chapter will elaborate on the concepts introduced here. Figure 6.2 shows the inheritance relationship between the class String and its superclass Object. A client that uses a String object is defined in Example 6.2. During the execution of the main() method, the String object created at (1) is denoted by two references: stringRef of the subclass String and objRef of the superclass Object. Walking through the code for the main() method reveals salient features of OOP.

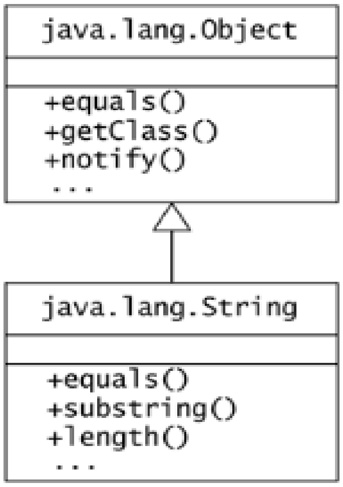
Example 6.2 Illustrating Inheritance

```
// String class is a subclass of Object class
class Client {
   public static void main(String[] args) {

      String stringRef = new String("Java");                    // (1)
      System.out.println("(2): " + stringRef.getClass());       // (2)
System.out.println("(3): " + stringRef.length());         // (3)
Object objRef = stringRef;                                // (4)
//   System.out.println("(5): " + objRef.length());           // (5) Not OK.
System.out.println("(6): " + objRef.equals("Java"));      // (6)
 System.out.println("(7): " + objRef.getClass());          // (7)
      stringRef = (String) objRef;                          // (8)
 System.out.println("(9): " + stringRef.equals("C++"));    // (9)
   }
}
```

Output from the program:

```
(2): class java.lang.String
(3): 4
(6): true
(7): class java.lang.String
(9): false
```

Figure 6.2. Inheritance Relationship between String and Object Classes



Inheriting from the Superclass

The subclass String inherits the method getClass() from the superclass Object. A client of the String class can directly invoke this inherited method on objects of the String class in the same way as if the method had been defined in the String class itself. In Example 6.2, this is illustrated at (2).

```
System.out.println("(2): " + stringRef.getClass());     // (2)
```

Extending the Superclass

The subclass String defines the method length(), which is not in the superclass Object, thereby extending the superclass. In Example 6.2, invocation of this new method on an object of class String is shown at (3).

```
System.out.println("(3): " + stringRef.length());       // (3)
```

Upcasting

A subclass reference can be assigned to a superclass reference because a subclass object can be used where a superclass object can be used. This is called upcasting, as references are assigned up the inheritance hierarchy. In Example 6.2, this is illustrated at (4), where the value of the subclass reference stringRef is assigned to the superclass reference objRef.

```
Object objRef = stringRef;                              // (4)
```

Both references denote the same String object after the assignment. One might be tempted to invoke methods exclusive to the String subclass via the superclass reference objRef, as illustrated at (5).

```
System.out.println("(5): " + objRef.length());          // (5) Not OK.
```

However, this will not work as the compiler does not know what object the reference objRef is denoting. It only knows the class of the reference. As the declaration of the Object class does not have a method called length(), this invocation of length() at (5) would be flagged as a compile-time error.

Method Overriding

In contrast to the situation at (5), the invocation of the equals() method at (6) using the superclass reference objRef is legal because the compiler can check that the Object class does define a method named equals.

```
System.out.println("(6): " + objRef.equals("Java"));    // (6)
```

Note that this method is redefined in the String class with the same signature (i.e., method name and parameters) and the same return type. This is called method overriding.

Polymorphism and Dynamic Method Binding

The invocation of the equals() method at (6), using the superclass reference objRef, does not necessarily invoke the equals() method from the Object class at runtime. The method invoked is dependent on the type of the actual object denoted by the reference at runtime. The actual method is determined by dynamic method lookup. The ability of a superclass reference to denote objects of its own class and its subclasses at runtime is called polymorphism.

Under normal program execution, the reference objRef will refer to an object of the String class at (6), resulting in the equals() method from the String class being executed, and not the one in the Object class.

The situation at (7), where the getClass() method is invoked using the superclass reference objRef, is allowed at compile time because the Object class defines a method named getClass.

```
    System.out.println("(7): " + objRef.getClass());        // (7)
```

In this case, under normal program execution, the reference objRef will refer to an object of the String class at (7). Dynamic method lookup determines which method implementation binds to the method signature getClass(). Since no getClass() method is defined in the String class, the method getClass() inherited from the Object class is thus executed.

Downcasting

Casting the value of a superclass reference to a subclass type is called downcasting. This is illustrated in Example 6.2 by assigning references down the inheritance hierarchy, which requires explicit casting.

stringRef = (String) objRef; // (8) System.out.println("(9): " + stringRef.equals("C++")); // (9) At (8), the source reference objRef is of type Object, which is the superclass of the class of the destination reference stringRef. If the reference objRef actually denoted an object of class String at runtime, the cast would convert it to the proper subclass type, so that the assignment to the reference stringRef would be legal at (8). The reference stringRef could then be used to invoke the equals() method on this String object, as at (9). Not surprisingly, the equals() method from the String class would be executed.

The compiler verifies that an inheritance relationship exists between the source reference type and the reference type specified in the cast. However, the cast can be invalid at runtime. If, at runtime, the reference objRef denotes an object of class Object or some unrelated subclass of class Object, then obviously casting the reference value to that of subclass String would be illegal. In such a case, a ClassCastException would be thrown at runtime. The instanceof operator can be used to determine the runtime type of an object before any cast is applied.

6.2 Overriding and Hiding Members Instance Method Overriding

Under certain circumstances, a subclass may override non-static methods defined in the superclass that would otherwise be inherited. When the method is invoked on an object of the subclass, it is the new method implementation in the subclass that is executed. The overridden method in the superclass is not inherited by the subclass, and the new method in the subclass must uphold the following rules of method overriding:

• The new method definition must have the same method signature (i.e., method name and parameters) and the same return type. Whether parameters in the overriding method should be final is at the discretion of the subclass. A method's signature does not encompass the final modifier of parameters, only their types and order. · The new method definition cannot narrow the accessibility of the method, but it can widen it.

• The new method definition can only specify all or none, or a subset of the exception classes (including their subclasses) specified in the throws clause of the overridden method in the superclass. These requirements also apply to interfaces, where a subinterface can override method prototypes from its superinterfaces.

In Example 6.3, the new definition of the getBill() method at (5) in the subclass TubeLight has the same signature and the same return type as the method at (2) in the superclass Light. The new definition specifies a subset of the exceptions (ZeroHoursException) thrown by the overridden method (exception class Invalid HoursException is a superclass of NegativeHoursException and ZeroHoursException). The new definition also widens the accessibility (public) from what it was in the overridden definition (protected). The overriding method also declares the parameter to be final. Invocation of the method getBill() on an object of subclass TubeLight using references of the subclass and the superclass at (12) and (13) respectively, results in the new definition at (5) being executed. Invocation of the method getBill() on an object of superclass Light using a reference of the superclass at (14), results in the overridden definition at (2) being executed.

Example 6.3 Overriding, Overloading, and Hiding

```
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
class ZeroHoursException extends InvalidHoursException {}
class Light {

   protected String billType = "Small bill";     // (1)
   protected double getBill(int noOfHours)
           throws InvalidHoursException {      // (2)

if (noOfHours < 0)

         throw new NegativeHoursException();
```

```
  double smallAmount = 10.0,
               smallBill = smallAmount * noOfHours;
System.out.println(billType + ": " + smallBill);
return smallBill;
    }

    public static void printBillType() {          // (3)
  System.out.println("Small bill");
    }

}

class TubeLight extends Light {

    public static String billType = "Large bill"; // (4) Hiding static field.
    public double getBill(final int noOfHours)
   throws ZeroHoursException {     // (5) Overriding instance method.
 if (noOfHours == 0)
            throw new ZeroHoursException();
        double largeAmount = 100.0,
largeBill = largeAmount * noOfHours;

        System.out.println(billType + ": " + largeBill);

 return largeBill;

    }

 public static void printBillType() {          // (6)
Hiding static method.
System.out.println(billType);
    }
 public double getBill() {                    // (7) Overloading method.
   System.out.println("No bill");
        return 0.0;
    }
}

public class Client {

    public static void main(String[] args)
            throws InvalidHoursException {        // (8)
        TubeLight tubeLight = new TubeLight();    // (9)
   Light     light1    = tubeLight;          // (10) Aliases.
  Light     light2    = new Light();        // (11)
       System.out.println("Invoke overridden instance method:");
  tubeLight.getBill(5);                    // (12) Invokes method at (5)

 light1.getBill(5);                        // (13) Invokes method at (5)

light2.getBill(5);                         // (14) Invokes method at (2)

   System.out.println("Access hidden field:");

 System.out.println(tubeLight.billType);   // (15) Accesses field at (4)
System.out.println(light1.billType);      // (16) Accesses field at (1)
System.out.println(light2.billType);      // (17) Accesses field at (1)

       System.out.println("Invoke hidden static method:");
 tubeLight.printBillType();               // (18) Invokes method at (6)
  light1.printBillType();                 // (19) Invokes method at (3)
 light2.printBillType();                  // (20) Invokes method at (3)
       System.out.println("Invoke overloaded method:");
tubeLight.getBill();                      // (21) Invokes method at (7)     }
}
```

Output from the program:

```
Invoke overridden instance method:
Large bill: 500.0
Large bill: 500.0
Small bill: 50.0
Access hidden field:
Large bill
Small bill
Small bill

 Invoke hidden static method:

Large bill
Small bill
Small bill
Invoke overloaded method:
No bill
```

A subclass must use the keyword super in order to invoke an overridden method in the superclass.

An instance method in a sublass cannot override a static method in the superclass. The compiler will flag this as an error. A static method is class-specific and not part of any object, while overriding methods are invoked on behalf of objects of the subclass. However, a static method in a subclass can hide a static method in the superclass (see below).

A final method cannot be overridden because the modifier final prevents method overriding. An attempt to override a final method will result in a compile-time error. However, an abstract method requires the non-abstract subclasses to override the method, in order to provide an implementation.

Accessibility modifier private for a method means that the method is not accessible outside the class in which it is defined; therefore, a subclass cannot override it. However, a subclass can give its own definition of such a method, which may have the same signature as the method in its superclass.

Field Hiding

A subclass cannot override fields of the superclass, but it can hide them. The subclass can define fields with the same name as in the superclass. If this is the case, the fields in the superclass cannot be accessed in the subclass by their simple names; therefore, they are not inherited by the subclass. Code in the subclass can use the keyword super to access such members, including hidden fields. A client can use a reference of the superclass to access members that are hidden in the subclass, as explained below. Of course, if the hidden field is static, it can also be accessed by the superclass name.

The following distinction between invoking instance methods on an object and accessing fields of an object must be noted. When an instance method is invoked on an object using a reference, it is the class of the current object denoted by the reference, not the type of the reference, that determines which method implementation will be executed. In Example 6.3 at (12), (13), and (14), this is evident from invoking the overridden method getBill(): the method from the class corresponding to the current object is executed, regardless of the reference type. When a field of an object is accessed using a reference, it is the type of the reference, not the class of the current object denoted by the reference, that determines which field will actually be accessed. In Example 6.3 at (15), (16), and (17), this is evident from accessing the hidden field billType: the field accessed is declared in the class corresponding to the reference type, regardless of the object denoted by the reference.

In contrast to method overriding where an instance method cannot override a static method, there are no such restrictions on the hiding of fields. The field billType is static in the subclass, but not in the superclass. The type of the fields need not be the same either, it is only the field name that matters in the hiding of fields.

Static Method Hiding

A static method cannot override an inherited instance method, but it can hide a static method if the exact requirements for overriding instance methods are fulfilled. A hidden superclass static method is not inherited. The compiler will flag an error if the signatures are the same but the other requirements regarding return type, throws clause, and accessibility are not met. If the signatures are different, the method name is overloaded, not hidden.

The binding of a method call to a method implementation is done at compile time if the method is static or final (private methods are implicitly final). Example 6.3 illustrates invocation of static methods. Analogous to accessing fields, the method invoked in (18), (19), and (20) is determined by the class of the reference. In (18) the class type is TubeLight, therefore, the static method printBillType() at (6) in this class is invoked. In (19) and (20) the class type is Light and the hidden static method printBillType() at (3) in that class is invoked. This is borne out by the output from the program.

A hidden static method can, of course, be invoked by using the superclass name in the subclass declaration. Additionally, the keyword super can be used in non-static code in the subclass declaration to invoke hidden static methods (see p. 238).

Overriding vs. Overloading

Method overriding should not be confused with method overloading. Method overriding requires the same method signature (name and parameters) and the same return type. Only non-final instance methods in the superclass that are directly accessible from the subclass are eligible for overriding. Overloading occurs when the method names are the same, but the parameter lists differ. Therefore, to overload methods, the parameters must differ in type, order, or number. As the return type is not a part of the signature, having different return types is not enough to overload methods.

A method can be overloaded in the class it is defined in or in a subclass of its class. Invoking an overridden method in the superclass from a subclass requires special syntax (e.g., the keyword super). This is not necessary for invoking an overloaded method in the superclass from a subclass. If the right kinds of arguments are passed in the method call occurring in the subclass, the overloaded method in the superclass will be invoked. In Example 6.3, the method getBill() at (2) in class Light is overridden in class TubeLight at (5) and overloaded at (7). When invoked at (21), the definition at (7) is executed.

Method Overloading Resolution

Example 6.4 illustrates how parameter resolution is done to choose the right implementation for an overloaded method. The method testIfOn() is overloaded at (1) and (2) in class OverloadResolution. The call client.testIfOn(tubeLight) at (3) satisfies the parameter lists in both the implementations given at (1) and (2), as the reference tubeLight, which denotes an object of class TubeLight, can also be assigned to a reference of its superclass Light. The most specific method, (2), is chosen, resulting in false being written on the terminal. The call client.testIfOn(light) at (4) only satisfies the parameter list in the implementation given at (1), resulting in true being written on the terminal.

Example 6.4 Overloaded Method Resolution

```
class Light { /* ... */ }

class TubeLight extends Light { /* ... */ }

public class OverloadResolution {
   boolean testIfOn(Light aLight)         { return true; }    // (1)
boolean testIfOn(TubeLight aTubeLight) { return false; }   // (2)
 public static void main(String[] args) {

        TubeLight tubeLight = new TubeLight();
        Light     light     = new Light();

        OverloadResolution client = new OverloadResolution();
System.out.println(client.testIfOn(tubeLight));// (3) ==> method at (2)
System.out.println(client.testIfOn(light));    // (4) ==> method at (1)

    }

}
```

Output from the program:

```
false
true
```

Object Reference super

The this reference is available in non-static code and refers to the current object. When an instance method is invoked, the this reference denotes the object on which the method is called. The keyword super can also be used in non-static code (e.g., in the body of an instance method), but only in a subclass, to access fields and invoke methods from the superclass. The keyword super provides a reference to the current object as an instance of its superclass. In method invocations with super, the method from the superclass is simply invoked regardless of the actual type of the object or whether the current class overrides the method. It is typically used to invoke methods that are overridden and to access members that are hidden in the subclass. Unlike the this keyword, the super keyword cannot be used as an ordinary reference. For example, it cannot be assigned to other references or cast to other reference types.

In Example 6.5, the method demonstrate() at (9) in class NeonLight makes use of the super keyword to access members higher up in its inheritance hierarchy. This is the case when the banner() method is invoked at (10). This method is defined at (4) in class Light and not in the immediate superclass of subclass NeonLight. The overridden method getBill() and its overloaded version at (6) and (8) in class TubeLight are invoked, using super at (12) and (11), respectively.

Class NeonLight is a subclass of class TubeLight, which is a subclass of class Light, which has a field named billType and a method named getBill defined at (1) and (2), respectively. One might be tempted to use the syntax super.super.getBill(20) in subclass NeonLight to invoke this method, but this is not a valid construct. One might also be tempted to cast the this reference to the class Light and try again as shown at (13). The output shows that the method getBill() at (6) in class TubeLight was executed, not the one from class Light. The reason is that a cast only changes the type of the reference (in this case to Light), not the class of the object (which is still NeonLight). Method invocation is determined by the class of the current object, resulting in the inherited method getBill() in class TubeLight being executed. There is no way to invoke the method getBill() in class Light from the subclass NeonLight.

At (14) the keyword super is used to access the field billType at (5) in class TubeLight. At (15) the field billType from class Light is accessed successfully by casting the this reference, because it is the type of the reference that determines the field accessed. From non-static code in a subclass, it is possible to directly access fields in a class higher up the inheritance hierarchy, by casting the this reference. However, it is futile to cast the this reference to invoke instance methods in a class higher up the inheritance hierarchy, as illustrated above in the case of the overridden method getBill().

Finally, calls to static methods at (16) and (17) using super and this references, exhibit runtime behavior analagous to accessing fields as discussed earlier.

Example 6.5 Using super Keyword

```
// Exceptions
class InvalidHoursException extends Exception {}
class NegativeHoursException extends InvalidHoursException {}
 class ZeroHoursException extends InvalidHoursException {}
class Light {

    protected String billType  = "Small bill";       // (1)
    protected double getBill(int noOfHours)
              throws InvalidHoursException {         // (2)
   if (noOfHours < 0)
            throw new NegativeHoursException();

double smallAmount = 10.0,
            smallBill = smallAmount * noOfHours;
System.out.println(billType + ": " + smallBill);
return smallBill;
   }

   public static void printBillType() {          // (3)
System.out.println("Small bill");
   }

   public void banner() {                        // (4)
 System.out.println("Let there be light!");     }


}

class TubeLight extends Light {

   public static String billType = "Large bill";    // (5) Hiding static field.
   public double getBill(final int noOfHours)
throws ZeroHoursException {       // (6) Overriding instance method.
if (noOfHours == 0)
          throw new ZeroHoursException();
double largeAmount = 100.0,
            largeBill = largeAmount * noOfHours;
 System.out.println(billType + ": " + largeBill);
return largeBill;
   }

   public static void printBillType() {          // (7) Hiding static method.
System.out.println(billType);
   }

   public double getBill() {                     // (8) Overloading method.
  System.out.println("No bill");
      return 0.0;
   }
}

class NeonLight extends TubeLight {

   // ...

   public void demonstrate()
```

```
            throws InvalidHoursException {           // (9)
        super.banner();                              // (10) Invokes method at (4)
  super.getBill();                              // (11) Invokes method at (8)
super.getBill(20);                              // (12) Invokes method at (6)
((Light) this).getBill(20);                     // (13) Invokes method at (6)
System.out.println(super.billType);         // (14) Accesses field at (5)
System.out.println(((Light) this).billType); // (15) Accesses field at (1)
 super.printBillType();                      // (16) Invokes method at (7)
  ((Light) this).printBillType();             // (17) Invokes method at (3)     }
}

public class Client {
    public static void main(String[] args)
                    throws InvalidHoursException {
  NeonLight neonRef = new NeonLight();
 neonRef.demonstrate();
    }
}
```

Output from the program:

```
Let there be light!
No bill
Large bill: 2000.0
Large bill: 2000.0
Large bill
Small bill
Large bill
Small bill
```

6.3 Chaining Constructors Using this() and super()

this() Constructor Call

Constructors cannot be inherited or overridden. They can be overloaded, but only in the same class. Since a constructor always has the same name as the class, each parameter list must be different when defining more than one constructor for a class. In Example 6.6, the class Light has three overloaded constructors. In the non-default constructor at (3), the this reference is used to access the fields shadowed by the parameters. In the main() method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call, as illustrated by the program output.

Example 6.6 Constructor Overloading

```
class Light {
   // Fields
   private int     noOfWatts;      // wattage
   private boolean indicator;      // on or off
 private String  location;       // placement     // Constructors
   Light() {                                    // (1) Explicit default constructor
   noOfWatts = 0;
      indicator = false;
      location  = "X";
      System.out.println("Returning from default constructor no. 1.");     }
   Light(int watts, boolean onOffState) {                      // (2) Non-default
noOfWatts = watts;
      indicator = onOffState;
      location  = "X";
      System.out.println("Returning from non-default constructor no. 2.");     }
   Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
this.noOfWatts = noOfWatts;

      this.indicator = indicator;

      this.location  = location;

      System.out.println("Returning from non-default constructor no. 3.");     }

}

  public class DemoConstructorCall {
   public static void main(String[] args) {                    // (4)
 System.out.println("Creating Light object no. 1.");
Light light1 = new Light();
      System.out.println("Creating Light object no. 2.");
Light light2 = new Light(250, true);
      System.out.println("Creating Light object no. 3.");
Light light3 = new Light(250, true, "attic");     }
}
```

Output from the program:

```
Creating Light object no. 1.
Returning from default constructor no. 1.
Creating Light object no. 2.
Returning from non-default constructor no. 2.
Creating Light object no. 3.
Returning from non-default constructor no. 3.
```

Example 6.7 illustrates the use of the this() construct, which is used to implement local chaining of constructors in the class when an instance of the class is created. The first two constructors at (1) and (2) from Example 6.6 have been rewritten using the this() construct. The this() construct can be regarded as being locally overloaded, since its parameters (and hence its signature) can vary, as shown in the body of the constructors at (1) and (2). The this() call invokes the constructor with the corresponding parameter list. In the main() method at (4), the appropriate constructor is invoked depending on the arguments in the constructor call when each of the three Light objects are created. Calling the default constructor to create a Light object results in the second and third constructors being executed as well.

This is confirmed by the output from the program. In this case, the output shows that the third constructor completed first, followed by the second, and finally the default constructor that was called first. Bearing in mind the definition of the constructors, the constructors are invoked in the reverse order; that is, invocation of the default constructor immediately leads to invocation of the second constructor by the call this(0, false), and its invocation leads to the third constructor being called immediately by the call this(watt, ind, "X"), with the completion of the execution in the reverse order of their invocation. Similarly, calling the second constructor to create an instance of the Light class results in the third constructor being executed as well.

Java requires that any this() call must occur as the first statement in a constructor. The this() call can be followed by any other relevant code. This restriction is due to Java's handling of constructor invocation in the superclass when an object of the subclass is created. This mechanism is explained in the next subsection.

Example 6.7 this() Constructor Call

```
class Light {

   // Fields
   private int     noOfWatts;
   private boolean indicator;
   private String  location;

   // Constructors
   Light() {                          // (1) Explicit default constructor
   this(0, false);
      System.out.println("Returning from default constructor no. 1.");     }
   Light(int watt, boolean ind) {        // (2) Non-default
this(watt, ind, "X");
      System.out.println("Returning from non-default constructor no. 2.");     }

   Light(int noOfWatts, boolean indicator, String location) {   // (3) Non-default

      this.noOfWatts = noOfWatts;
```

```
        this.indicator = indicator;
        this.location  = location;
        System.out.println("Returning from non-default constructor no. 3.");     }
}

public class DemoThisCall {
    public static void main(String[] args) {                      // (4)
 System.out.println("Creating Light object no. 1.");
 Light light1 = new Light();                              // (5)
System.out.println("Creating Light object no. 2.");
 Light light2 = new Light(250, true);                     // (6)
System.out.println("Creating Light object no. 3.");
  Light light3 = new Light(250, true, "attic");           // (7)     }
}
```

```
Output from the program:
Creating Light object no. 1.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Returning from default constructor no. 1.
Creating Light object no. 2.
Returning from non-default constructor no. 3.
Returning from non-default constructor no. 2.
Creating Light object no. 3.
Returning from non-default constructor no. 3.
```

super() Constructor Call

The super() construct is used in a subclass constructor to invoke a constructor in the immediate superclass. This allows the subclass to influence the initialization of its inherited state when an object of the subclass is created. A super() call in the constructor of a subclass will result in the execution of the relevant constructor from the superclass, based on the signature of the call. Since the superclass name is known in the subclass declaration, the superclass constructor invoked is determined by the signature of the parameter list.

A constructor in a subclass can access the class's inherited members directly (i.e., by their simple name). The keyword super can also be used in a subclass constructor to access inherited members via its superclass. One might be tempted to use the super keyword in a constructor to specify initial values of inherited fields. However, the super() construct provides a better solution, using superclass constructors to initialize the inherited state.

In Example 6.8, the non-default constructor at (3) of the class Light has a super() call (with no arguments) at (4). Although the constructor is not strictly necessary, as the compiler will insert one—as explained below—it is included for expositional purposes. The non-default constructor at

(6) of class TubeLight has a super() call (with three arguments) at (7). This super() call will match the non-default constructor at (3) of superclass Light. This is evident from the program output.

Example 6.8 super() Constructor Call

```
class Light {
   // Fields
   private int     noOfWatts;
   private boolean indicator;
   private String  location;

   // Constructors
   Light() {                          // (1) Explicit default constructor
    this(0, false);
      System.out.println(
         "Returning from default constructor no. 1 in class Light");     }
   Light(int watt, boolean ind) {                         // (2) Non-default
 this(watt, ind, "X");
      System.out.println(
         "Returning from non-default constructor no. 2 in class Light");     }
   Light(int noOfWatts, boolean indicator, String location) {  // (3) Non-default
 super();                                          // (4)
this.noOfWatts = noOfWatts;
      this.indicator = indicator;

      this.location  = location;

      System.out.println(

         "Returning from non-default constructor no. 3 in class Light");     }

}

  class TubeLight extends Light {
   // Instance variables
   private int tubeLength;
   private int colorNo;

   TubeLight(int tubeLength, int colorNo) {                   // (5) Non-default
this(tubeLength, colorNo, 100, true, "Unknown");
 System.out.println(
         "Returning from non-default constructor no. 1 in class TubeLight");     }
   TubeLight(int tubeLength, int colorNo, int noOfWatts,
  boolean indicator, String location) {              // (6) Non-default
super(noOfWatts, indicator, location);                   // (7)
 this.tubeLength = tubeLength;
       this.colorNo    = colorNo;
       System.out.println(
          "Returning from non-default constructor no. 2 in class TubeLight");     }
}
public class Chaining {
   public static void main(String[] args) {

       System.out.println("Creating a TubeLight object.");

TubeLight tubeLightRef = new TubeLight(20, 5);          // (8)     }

}
```

Output from the program:

```
Creating a TubeLight object.
Returning from non-default constructor no. 3 in class Light
Returning from non-default constructor no. 2 in class TubeLight
Returning from non-default constructor no. 1 in class TubeLight
```

The super() construct has the same restrictions as the this() construct: if used, the super() call must occur as the first statement in a constructor, and it can only be used in a constructor declaration. This implies that this() and super() calls cannot both occur in the same constructor. The this() construct is used to chain constructors in the same class, and the constructor at the end of such a chain can invoke a superclass constructor using the super() construct. Just as the this() construct leads to chaining of constructors in the same class, the super() construct leads to chaining of subclass constructors to superclass constructors. This chaining behavior guarantees that all superclass constructors are called, starting with the constructor of the class being instantiated, all the way to the top of the inheritance hierarchy, which is always the Object class. Note that the body of the constructors is executed in the reverse order to the call order, as super() can only occur as the first statement in a constructor. This ensures that the constructor from the Object class is completed first, followed by the constructors in the other classes down to the class being instantiated in the inheritance hierarchy. This is called (subclass-superclass) constructor chaining. The output from Example 6.8 clearly illustrates this chain of events when an object of class TubeLight is created.

If a constructor at the end of a this()-chain (which may not be a chain at all if no this() call is invoked) does not have an explicit call to super(), then the call super() (without the parameters) is implicitly inserted to invoke the default constructor of the superclass. In other words, if a constructor has neither a this() nor a super() call as its first statement, then a super() call to the default constructor in the superclass is inserted. The code

```
class A {
   public A() {}
   // ...
}
class B extends A {
   // no constructors
   // ...
}

class A {
   public A() { super(); }      // (1)
   // ...
}
class B extends A {
   public B() { super(); }      // (2)
   // ...
}
```

is equivalent to

where the default constructors with calls to the default superclass constructor are inserted in the code.

If a class only defines non-default constructors (i.e., only constructors with parameters), then its subclasses cannot rely on the implicit super() call being inserted. This will be flagged as a compile-time error. The subclasses must then explicitly call a superclass constructor, using the super() construct with the right arguments. class NeonLight extends TubeLight {

```
   // Field
   String sign;

   NeonLight() {                        // (1)
      super(10, 2, 100, true, "Roof-top");    // (2) Cannot be commented out.
 sign = "All will be revealed!";
   }
   // ...

}
```

The previous declaration of the subclass NeonLight provides a constructor at (1). The call at (2) of the constructor in the superclass TubeLight cannot be omitted. If it is omitted, any insertion of a super() call (with no arguments) in this constructor will not match any default constructor in the superclass TubeLight, as this superclass does not provide one. The superclass TubeLight only provides non-default constructors. The class NeonLight will not compile unless an explicit super() call (with valid arguments) is inserted at (2).

If the superclass provides non-default constructors only (i.e., does not have a default constructor), then this has implications for its subclasses. A subclass that relies on its implicit default constructor will fail to compile. This is because the implicit default constructor of the subclass will attempt to call the non-existent default constructor in the superclass. Any constructor in a subclass must explicitly use the super() call, with the appropriate arguments, to invoke a non-default constructor in the superclass. This is because the constructor in the subclass cannot rely on an implicit super() call to the default constructor in the superclass.

6.4 Interfaces

Extending classes using single implementation inheritance creates new class types. A superclass reference can denote objects of its own type and its subclasses strictly according to the inheritance hierarchy. Because this relationship is linear, it rules out multiple implementation inheritance, that is, a subclass inheriting from more than one superclass. Instead Java provides interfaces, which not only allow new named reference types to be introduced, but also permit multiple interface inheritance.

Defining Interfaces

A top-level interface has the following general syntax:

```
<accessibility modifier> interface <interface name>
  <extends interface clause> // Interface header
{ // Interface body
     <constant declarations>
     <method prototype declarations>
     <nested class declarations>
     <nested interface declarations>
   }
```

In the interface header, the name of the interface is preceded by the keyword interface. In addition, the interface header can specify the following information:

- scope or accessibility modifier
- any interfaces it extends

The interface body can contain member declarations which comprise

- constant declarations
- method prototype declarations
- nested class and interface declarations

An interface does not provide any implementation and is, therefore, abstract by definition. This means that it cannot be instantiated, but classes can implement it by providing implementations for its method prototypes. Declaring an interface abstract is superfluous and seldom done.

The member declarations can appear in any order in the interface body. Since interfaces are meant to be implemented by classes, interface members implicitly have public accessibility and the public modifier is omitted.

Interfaces with empty bodies are often used as markers to tag classes as having a certain property or behavior. Such interfaces are also called ability interfaces. Java APIs provide several examples of such marker interfaces: java.lang.Cloneable, java.io.Serializable, java.util.EventListener.

Method Prototype Declarations

An interface defines a contract by specifying a set of method prototypes, but no implementation. The methods in an interface are all implicitly abstract and public by virtue of their definition. A method prototype has the same syntax as an abstract method. However, only the modifiers abstract and public are allowed, but these are invariably omitted.

<return type> <method name> (<parameter list>) <throws clause>;

Example 6.9 declares two interfaces: IStack at (1) and ISafeStack at (5). These interfaces are discussed in the subsequent subsections.

Example 6.9 Interfaces

```
interface IStack {                              // (1)
void   push(Object item);
   Object pop();
}

class StackImpl implements IStack {                   // (2)
protected Object[] stackArray;
   protected int      tos;  // top of stack

   public StackImpl(int capacity) {

      stackArray = new Object[capacity];

      tos       = -1;
   }

   public void push(Object item)                          // (3)
 { stackArray[++tos] = item; }

   public Object pop() {                                  // (4)
Object objRef = stackArray[tos];
      stackArray[tos] = null;
      tos--;
      return objRef;
   }

   public Object peek() { return stackArray[tos]; } }

interface ISafeStack extends IStack {                     // (5)

   boolean isEmpty();
   boolean isFull();
}

class SafeStackImpl extends StackImpl implements ISafeStack {     // (6)

   public SafeStackImpl(int capacity) { super(capacity); }

public boolean isEmpty() { return tos < 0; }              // (7)

public boolean isFull() { return tos >= stackArray.length-1; }// (8) }

public class StackUser {


   public static void main(String[] args) {               // (9)
SafeStackImpl safeStackRef  = new SafeStackImpl(10);
StackImpl     stackRef      = safeStackRef;
  ISafeStack    isafeStackRef = safeStackRef;
IStack        istackRef     = safeStackRef;
Object        objRef        = safeStackRef;
      safeStackRef.push("Dollars");                       // (10)
 stackRef.push("Kroner");
      System.out.println(isafeStackRef.pop());
System.out.println(istackRef.pop());
      System.out.println(objRef.getClass());     }

}
```

Output from the program:

```
Kroner
Dollars
class SafeStackImpl
```

Implementing Interfaces

Any class can elect to implement, wholly or partially, zero or more interfaces. A class specifies the interfaces it implements as a comma-separated list of unique interface names in an implements clause in the class header. The interface methods must all have public accessibility when implemented in the class (or its subclasses). A class can neither narrow the accessibility of an interface method nor specify new exceptions in the method's throws clause, as attempting to do so would amount to altering the interface's contract, which is illegal. The criteria for overriding methods also apply when implementing interface methods.

A class can provide implementations of methods declared in an interface, but it does not reap the benefits of interfaces unless the interface name is explicitly specified in its implements clause.

In Example 6.9, the class StackImpl implements the interface IStack by both specifying the interface name using the implements clause in its class header at (2) and providing the implementation for the methods in the interface at (3) and (4). Changing the public accessibility of these methods in the class will result in a compile-time error, as this would narrow their accessibility.

A class can choose to implement only some of the methods of its interfaces, (i.e., give a partial implementation of its interfaces). The class must then be declared as abstract. Note that interface methods cannot be declared static, because they comprise the contract fulfilled by the objects of the class implementing the interface. Interface methods are always implemented as instance methods.

The interfaces a class implements and the classes it extends (directly or indirectly) are called supertypes of the class. Conversely, the class is a subtype of its supertypes. Classes implementing interfaces introduce multiple interface inheritance into their linear implementation inheritance hierarchy. However, note that regardless of how many interfaces a class implements directly or indirectly, it only provides a single implementation of a member that might have multiple declarations in the interfaces.

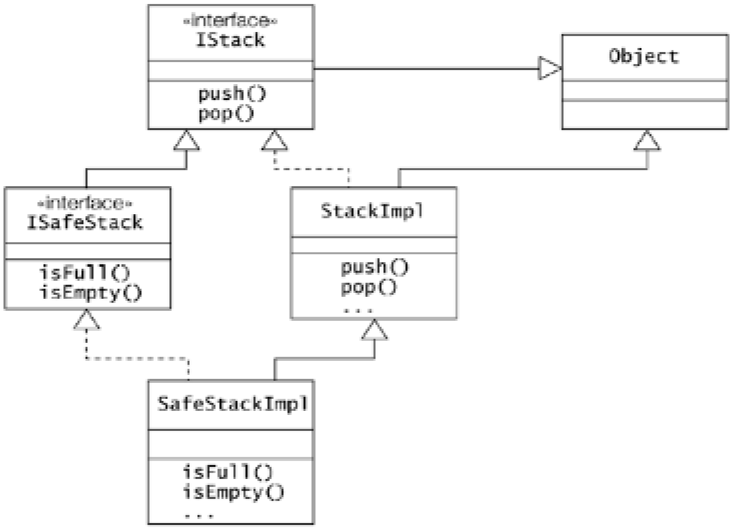
Extending Interfaces

An interface can extend other interfaces, using the extends clause. Unlike extending classes, an interface can extend several interfaces. The interfaces extended by an interface (directly or indirectly), are called superinterfaces. Conversely, the interface is a subinterface of its superinterfaces. Since interfaces define new reference types, superinterfaces and subinterfaces are also supertypes and subtypes, respectively.

A subinterface inherits all methods from its superinterfaces, as their method declarations are all implicitly public. A subinterface can override method prototype declarations from its superinterfaces. Overridden methods are not inherited. Method prototype declarations can also be overloaded, analogous to method overloading in classes.

Example 6.9 provides an example of multiple inheritance in Java. In Example 6.9, the interface ISafeStack extends the interface IStack at (5). The class SafeStackImpl both extends the StackImpl class and implements the ISafeStack interface at (6). Both the implementation and interface inheritance hierarchies for classes and interfaces defined in Example 6.9 are shown in Figure 6.3.

Figure 6.3. Inheritance Relations



In UML, an interface resembles a class. One way to differentiate between them is to use an ・interface・ stereotype as in Figure 6.3. Interface inheritance is shown similar to implementation inheritance, but with a dotted inheritance arrow. Thinking in terms of types, every reference type in Java is a subtype of Object type. This means that any interface type is also a subtype of Object type. We have augmented Figure 6.3 with an extra inheritance arrow to show this subtype relation.

It is instructive to note how class SafeStackImpl implements the ISafeStack interface: it inherits implementations of the push() and pop() methods from its superclass StackImpl, and provides its own implementation of the isFull() and isEmpty() methods from the ISafeStack interface. The interface ISafeStack inherits two method prototypes from its superinterface IStack. All its methods are implemented by the SafeStackImpl class. The class SafeStackImpl implicitly implements the IStack interface: it implements the ISafeStack interface that inherits from the IStack interface. This is readily evident from the diamond shape of the inheritance hierarchy in Figure 6.3. There is only one single implementation inheritance into the class SafeStackImpl.

Note that there are three different inheritance relations at work when defining inheritance among classes and interfaces:

1. Linear implementation inheritance hierarchy between classes: a class extends another class (subclasses–superclasses).
2. Multiple inheritance hierarchy between interfaces: an interface extends other interfaces (subinterfaces–superinterfaces).
3. Multiple interface inheritance hierarchy between interfaces and classes: a class implements interfaces.

Although interfaces cannot be instantiated, references of an interface type can be declared. References to objects of a class can be assigned to references of the class' supertypes. In Example 6.9, an object of the class SafeStackImpl is created in the main() method of the class StackUser at (9). The reference value of the object is assigned to references of all the object's supertypes, which are used to manipulate the object.

Constants in Interfaces

An interface can also define named constants. Such constants are defined by field declarations and are considered to be public, static and final. These modifiers are usually omitted from the declaration. Such a constant must be initialized with an initializer expression.

An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly without using the fully qualified name. Such a client inherits the interface constants. Typical usage of constants in interfaces is illustrated in Example 6.10, showing both direct access and use of fully qualified names at (1) and (2), respectively.

Extending an interface that has constants is analogous to extending a class having static variables. In particular, these constants can be hidden by the subinterfaces. In the case of multiple inheritance of interface constants, any name conflicts can be resolved using fully qualified names for the constants involved.

Example 6.10 Variables in Interfaces

```
interface Constants {
   double PI_APPROXIMATION = 3.14;
   String AREA_UNITS = " sq.cm.";
   String LENGTH_UNITS = " cm.";
}

public class Client implements Constants {
    public static void main(String[] args) {
        double radius = 1.5;
        System.out.println("Area of circle is " +
(PI_APPROXIMATION*radius*radius) +
AREA_UNITS);             // (1) Direct access.
System.out.println("Circumference of circle is " +
(2*Constants.PI_APPROXIMATION*radius) +
  Constants.LENGTH_UNITS); // (2) Fully qualified name.     }
}
```

Output from the program:

```
Area of circle is 7.0649999999999995 sq.cm.
 Circumference of circle is 9.42 cm.
```

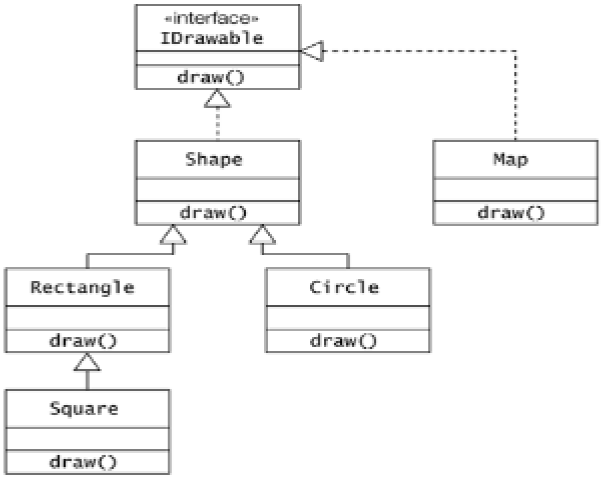
6.5 Polymorphism and Dynamic Method Lookup

Which object a reference will actually denote during runtime, cannot always be determined at compile time. Polymorphism allows a reference to denote objects of different types at different times during execution. A supertype reference exhibits polymorphic behavior, since it can denote objects of its subtypes.

When a non-private instance method is invoked on an object, the method definition actually executed is determined both by the type of the object at runtime and the method signature. Dynamic method lookup is the process of determining which method definition a method signature denotes during runtime, based on the type of the object. However, a call to a private instance method is not polymorphic. Such a call can only occur within the class, and gets bound to the private method implementation at compile time.

The inheritance hierarchy depicted in Figure 6.5 is implemented in Example 6.11. The implementation of the method draw() is overridden in all subclasses of the class Shape. The invocation of the draw() method in the two loops at (3) and (4) in Example 6.11, relies on the polymorphic behavior of references and dynamic method lookup. The array shapes holds Shape references denoting a Circle, a Rectangle and a Square, as shown at (1). At runtime, dynamic lookup determines the draw() implementation to execute, based on the type of the object denoted by each element in the array. This is also the case for the elements of the array drawables at (2), which holds IDrawable references that can be assigned any object of a class that implements the IDrawable interface. The first loop will still work without any change if objects of new subclasses of the class Shape are added to the array shapes. If they did not override the draw() method, then an inherited version of the method would be executed. This polymorphic behavior applies to the array drawables, where the subtype objects are guaranteed to have implemented the IDrawable interface.

Figure 6.5. Polymorphic Methods



Polymorphism and dynamic method lookup form a powerful programming paradigm that simplifies client definitions, encourages object decoupling, and supports dynamically changing relationships between objects at runtime.

Example 6.11 Polymorphism and Dynamic Method Lookup

```
interface IDrawable {
   void draw();
}
class Shape implements IDrawable {
   public void draw() { System.out.println("Drawing a Shape."); } }
class Circle extends Shape {
   public void draw() { System.out.println("Drawing a Circle."); } }
class Rectangle extends Shape {
   public void draw() { System.out.println("Drawing a Rectangle."); } }
class Square extends Rectangle {
   public void draw() { System.out.println("Drawing a Square."); } }
class Map implements IDrawable {
   public void draw() { System.out.println("Drawing a Map."); } }
public class PolymorphRefs {

   public static void main(String[] args) {
      Shape[] shapes = {new Circle(), new Rectangle(), new Square()};   // (1)
 IDrawable[] drawables = {new Shape(), new Rectangle(), new Map()};// (2)

      System.out.println("Draw shapes:");
```

```
        for (int i = 0; i < shapes.length; i++)                      // (3)
  shapes[i].draw();

        System.out.println("Draw drawables:");
for (int i = 0; i < drawables.length; i++)                        // (4)
drawables[i].draw();
   }
}
```

Output from the program:

```
Draw shapes:
Drawing a Circle.
Drawing a Rectangle.
Drawing a Square.
Draw drawables:
Drawing a Shape.
Drawing a Rectangle.
Drawing a Map.
```

07

Nested Classes & Interfaces

7.1 Overview of Nested Classes and Interfaces

A class that is declared within another class or interface, is called a nested class. Similarly, an interface that is declared within another class or interface, is called a nested interface. A top-level class or a top-level interface is one that is not nested.

In addition to the top-level classes and interfaces, there are four categories of nested classes and one of nested interfaces, defined by the context these classes and interfaces are declared in:

- static member classes and interfaces
- non-static member classes
- local classes
- anonymous classes

The last three categories are collectively known as inner classes. They differ from non-inner classes in one important aspect: that an instance of an inner class may be associated with an instance of the enclosing class. The instance of the enclosing class is called the immediately enclosing instance. An instance of an inner class can access the members of its immediately enclosing instance by their simple name.

A static member class or interface is defined as a static member in a class or an interface. Such a nested class can be instantiated like any ordinary top-level class, using its full name. No enclosing instance is required to instantiate a static member class. Note that there are no non-static member, local, or anonymous interfaces. Interfaces are always defined either at the top level or as static members.

Non-static member classes are defined as instance members of other classes, just like fields and instance methods are defined in a class. An instance of a non-static member class always has an enclosing instance associated with it.

Local classes can be defined in the context of a block as in a method body or a local block, just as local variables can be defined in a method body or a local block.

Anonymous classes can be defined as expressions and instantiated on the fly. An instance of a local (or an anonymous) class has an enclosing instance associated with it, if the local (or anonymous) class is declared in a non-static context.

A nested class or interface cannot have the same name as any of its enclosing classes or interfaces.

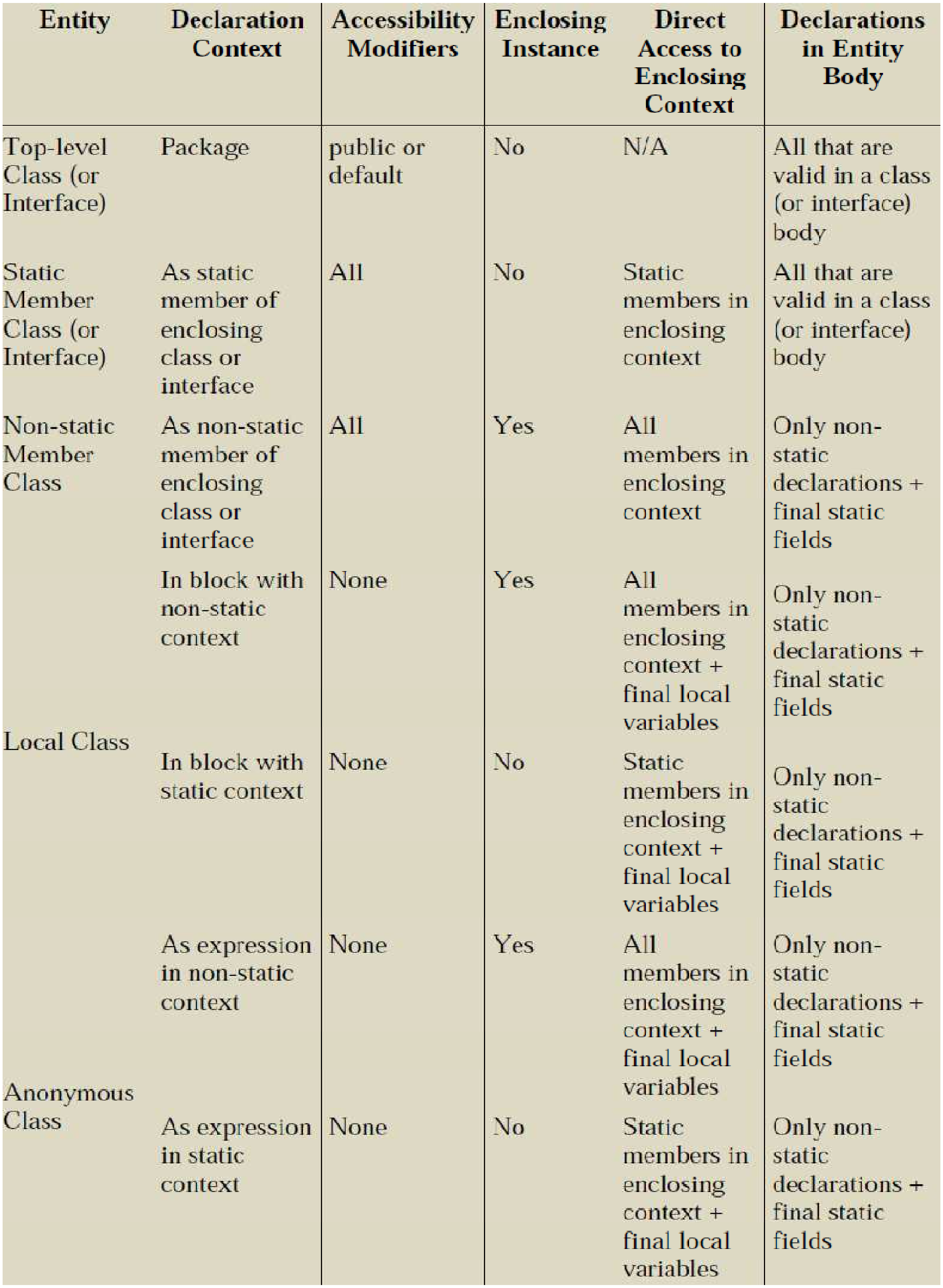
Skeletal code for nested classes is shown Example 7.1. Table 7.1 presents a summary of various aspects relating to nested classes and interfaces. The Entity column lists the different kinds of classes and interfaces that can be declared. The Declaration Context column lists the lexical context in which the class or interface can be defined. The Accessibility Modifiers column indicates what accessibility can be specified for the class or interface. The Enclosing Instance column specifies whether an enclosing instance is associated with an instance of the class. The Direct Access to Enclosing Context column lists what is directly accessible in the enclosing context from within the class or interface. The Declarations in Entity Body column refers to what can be declared in the class or interface body. Subsequent sections on each nested class elaborate on the summary presented in Table 7.1. (N/A in the table means not applicable.)

Example 7.1 Overview of Nested Classes and Interfaces

```
class TLC {                  // (1) Top level class
   static class SMC {/*...*/} // (2) Static member class
   interface SMI {/*...*/}    // (3) Static member interface
   class NSMC {/*...*/}       // (4) Non-static member (inner) class
   void nsm() {
      class NSLC {/*...*/}   // (5) Local (inner) class in non-static context     }

   static void sm() {
      class SLC {/*...*/}    // (6) Local (inner) class in static context     }

   SMC nsf = new SMC()
         {/*...*/};     // (7) Annonymous (inner) class in non-static context
   static SMI sf = new SMI()
         {/*...*/};     // (8) Annonymous (inner) class in static context }
```

Nested type (i.e., nested classes and interfaces) can be regarded as a form of encapsulation, enforcing relationships between types by greater proximity. They allow structuring of types and a special binding relationship between a nested object and its enclosing instance. Used judiciously, they can be beneficial, but unrestrained use of nested classes can easily result in unreadable code.

Table 7.1. Overview of Classes and Interfaces



Entity	Declaration Context	Accessibility Modifiers	Enclosing Instance	Direct Access to Enclosing Context	Declarations in Entity Body
Top-level Class (or Interface)	Package	public or default	No	N/A	All that are valid in a class (or interface) body
Static Member Class (or Interface)	As static member of enclosing class or interface	All	No	Static members in enclosing context	All that are valid in a class (or interface) body
Non-static Member Class	As non-static member of enclosing class or interface	All	Yes	All members in enclosing context	Only non-static declarations + final static fields
Local Class	In block with non-static context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields
	In block with static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields
Anonymous Class	As expression in non-static context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields
	As expression in static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields

7.2 Static Member Classes and Interfaces

Declaring and Using Static Member Classes and Interfaces

A static member class or a static member interface comprises the same declarations as those allowed in an ordinary top-level class or interface. A static member class must be declared as a static member of an enclosing class or interface. Nested interfaces are considered implicitly static, the keyword static can, therefore, be omitted. Since static member classes and interfaces are members of an enclosing class or interface, they can have any member accessibility.

Static member classes and interfaces can only be nested within other static member or top-level classes and interfaces.

In Example 7.2, the top-level class TopLevelClass at (1) contains a static member class StaticMemberClass_1 at (2), which in turn defines a static member interface StaticMemberInterface_1_1 at (3) and a static member class StaticMemberClass_1_1 at (4). The static member class StaticMemberClass_1_1 at (4) implements the static member interface StaticMemberInterface_1 at (5). Note that each static member class is defined as static, just like static variables and methods in a top-level class.

Example 7.2 Static Member Classes and Interfaces

```
 // Filename: TopLevelClass.java
package express;

public class TopLevelClass {                        // (1)     // ...
   public static class StaticMemberClass_1 {         // (2)        // ...
      private interface StaticMemberInterface_1_1 { // (3)            // ...
      }
      public static class StaticMemberClass_1_1
  implements StaticMemberInterface_1 {      // (4)            // ...
      }
   }
   interface StaticMemberInterface_1
      extends StaticMemberClass_1.StaticMemberInterface_1_1 {       // (5)        // ...
   }
}

class AnotherTopLevelClass
      implements TopLevelClass.StaticMemberInterface_1 {            // (6)
   TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1   objRef1   =                    new
TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1(); // (7)

   //TopLevelClass.StaticMemberClass_1.StaticMemberInterface_1_1 ref; // (8)
```

// ...}

The full name of a (static or non-static) member class or interface includes the names of the classes and interfaces it is lexically nested in. For example, the full name of the member class StaticMemberClass_1_1 at (4) is TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1. The full name of the member interface StaticMemberInterface_1 at (5) is TopLevelClass.StaticMemberInterface_1. Each member class or interface is uniquely identified by this naming syntax, which is a generalization of the naming scheme for packages. The full name can be used in exactly the same way as any other top-level class or interface name, as shown at (6) and (7). Such a member's fully qualified name is its full name prefixed by the name of its package. For example, the fully qualified name of the member class at (4) is express.TopLevelClass.StaticMemberClass_1.StaticMemberClass_1_1. Note that a member class or interface, cannot have the same name as an enclosing class, interface, or package.

For all intents and purposes, a static member class or interface is very much like any other top-level class or interface. Static variables and methods belong to a class, and not to instances of the class. The same is true for static member classes and interfaces.

Within the scope of its top-level class or interface, a member class or interface can be referenced regardless of its accessibility modifier and lexical nesting, as shown at (5) in Example 7.2. Its accessibility modifier (and that of the types making up its full name) come into play when it is referenced in an external client. The declaration at (8) in Example 7.2 will not compile because the member interface TopLevelClass.StaticMemberClass_1.StaticMemberInterface_1_1 has private accessibility.

A static member class can be instantiated without any reference to any instance of the enclosing context, as is the case for instantiating top-level classes. An example of creating an instance of a static member class is shown at (7) in Example 7.2, using the new operator.

If the file TopLevelClass.java containing the declarations in Example 7.2 is compiled, it will result in the generation of the following class files, where each file corresponds to a class or interface definition:

TopLevelClass$StaticMemberClass_1$StaticMemberClass_1_1.class
TopLevelClass$StaticMemberClass_1$StaticMemberInterface_1_1.class
TopLevelClass$StaticMemberClass_1.class TopLevelClass$StaticMemberInterface_1.class
TopLevelClass.class
AnotherTopLevelClass.class

Note how the full class name corresponds to the class file name (minus the extension), with the dollar sign ($) replaced by the dot sign (.).

There is seldom any reason to import member classes or interfaces from packages. It would undermine the encapsulation achieved by such classes or interfaces. However, a compilation unit for a named package can use the import declaration to provide a short cut for the names of member classes and interfaces. (Java does not allow imports from the unnamed package.) Some variations on usage of the import declaration for member classes are shown in Example 7.3.

Example 7.3 Importing Member Classes

```
// Filename: Client1.java
package express;
import express.TopLevelClass.*;                         // (1)
public class Client1 {
   StaticMemberClass_1.StaticMemberClass_1_1 objRef1 =
new StaticMemberClass_1.StaticMemberClass_1_1();     // (2) }
                                                              // Filename:
Client2.java
package express;
import express.TopLevelClass.StaticMemberClass_1.*;        // (3)
public class Client2 {
   StaticMemberClass_1_1 objRef2 = new StaticMemberClass_1_1(); // (4) }
class SomeClass implements
   express.TopLevelClass.StaticMemberInterface_1 {          // (5)    /* ... */
}
```

In the file Client1.java, the import statement at (1) allows the static member class StaticMemberClass_1_1 to be referenced as StaticMemberClass_1.StaticMemberClass_1_1 as at (2), whereas in the file Client2.java, the import statement at (3) will allow the same class to be referenced using its simple name, as at (4). At (5) the fully qualified name of the static member interface is used in an implements clause.

Accessing Members in Enclosing Context

Static code does not have a this reference and can, therefore, only directly access other members that are declared static within the same class. Since static member classes are static, they do not have any notion of an enclosing instance of the enclosing context. This means that any code in a static member class can only directly access static members in its enclosing context, not instance members in its enclosing context.

Figure 7.1 is a class diagram that illustrates static member classes and interfaces. These are shown as members of the enclosing context, with the {static} tag to indicate that they are static, that is, they can be instantiated without regard to any object of the enclosing context. Since they are members of a class or an interface, their accessibility can be specified exactly like that of any other member of a class or interface. The classes from the diagram are implemented in Example 7.4.

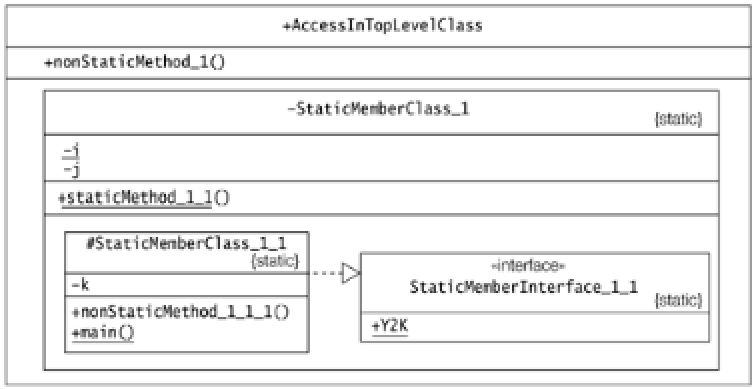
Example 7.4 Accessing Members in Enclosing Context

```
// Filename: AccessInTopLevelClass.java
public class AccessInTopLevelClass {                          // (1)
    public void nonStaticMethod_1() {                         // (2)
System.out.println("nonstaticMethod_1 in AccessInTopLevelClass");     }

    private static class StaticMemberClass_1 {                // (3)
 private static int i;                       // (4)        private int j;                                // (5)
        public static void staticMethod_1_1() {               // (6)
System.out.println("staticMethod_1_1 in StaticMemberClass_1");        }

        interface StaticMemberInterface_1_1 { int Y2K = 2000; } // (7)
        protected static class StaticMemberClass_1_1
implements StaticMemberInterface_1_1 {          // (8)
            private int k = Y2K;                          // (9)

public void nonStaticMethod_1_1_1() {               // (10)
   //  int jj = j;              // (11) Not OK.
  int ii = i;              // (12)
                int kk = k;              // (13)

            //  nonStaticMethod_1();     // (14) Not OK.
   staticMethod_1_1();      // (15)
            }

            public static void main (String[] args) {
int ii = i;              // (16)
            //  int kk = k;              // (17) Not OK.
    staticMethod_1_1();      // (18)
            }
        }
    }
}
```

Output from the program:

```
staticMethod_1_1 in StaticMemberClass_1
```

Figure 7.1. Static Member Classes and Interfaces



Example 7.4 demonstrates accessing members directly in the enclosing context of class StaticMemberClass_1_1 defined at (8). The initialization of the field k at (9) is valid, since the field Y2K, defined in the outer interface StaticMemberInterface_1_1 at (7), is implicitly static. The compiler will flag an error at (11) and (14) in the method nonStaticMethod_1_1_1(), because direct access to non-static members in the enclosing class is not permitted by any method in a static member class. It will also flag an error at (17) in the method main(), because a static method cannot access directly other non-static fields in its own class. Statements at (16) and (18) access static members only in the enclosing context. The references in these statements can also be specified using full names.

```
 int ii = AccessInTopLevelClass.StaticMemberClass_1.i;
AccessInTopLevelClass.StaticMemberClass_1.staticMethod_1_1();
```

Note that a static member class can define both static and instance members, like any other top-level class. However, its code can only directly access static members in its enclosing context.

A static member class, being a member of the enclosing class or interface, can have any accessibility (public, protected, package/default, private), like any other member of a class. The class StaticMemberClass_1 at (3) has private accessibility, whereas its nested class StaticMemberClass_1_1 at (8) has protected accessibility. The class StaticMemberClass_1_1 defines the method main(), which can be executed by the command

```
>java AccessInTopLevelClass$StaticMemberClass_1$StaticMemberClass_1_1
```

Note that the class StaticMemberClass_1_1 is specified using the full name of the class file, minus the extension.

7.3 Non-static Member Classes

Non-static member classes are inner classes that are defined without the keyword static, as members of an enclosing class or interface. Non-static member classes are on par with other non-static members defined in a class. The following aspects about non-static member classes should be noted:

· An instance of a non-static member class can only exist with an instance of its enclosing class. This means that an instance of a non-static member class must be created in the context of an instance of the enclosing class. This also means that a non-static member class cannot have static members. In other words, the non-static member class does not provide any services, only instances of the class do. However, final static variables are allowed, as these are constants.

· Code in a non-static member class can directly refer to any member (including nested) of any enclosing class or interface, including private members. No explicit reference is required.

• Since a non-static member class is a member of an enclosing class, it can have any accessibility: public, package/default, protected, or private.

A typical application of non-static member classes is implementing data structures. Code below outlines implementing a linked list, where the Node class is nested in the LinkedList class. Since the non-static member class Node is declared private, it is not accessible outside of class LinkedList. Nesting promotes encapsulation, and the close proximity allows classes to exploit each others capabilities.

```
                   class LinkedList {                                      // (1)
                    private class Node {                                  // (2)
                             private Object data;    // Data
    private Node next;      // Next node
    // ...
  }

  protected Node head;
  protected Node tail;
  // ...
}
```

Instantiating Non-static Member Classes

In Example 7.5, the class ToplevelClass at (1) defines a non-static member class at (5). Declaration of a static variable at (6) in class NonStaticMemberClass is flagged as a compile-time error, but defining a final static variable at (7) is allowed.

Example 7.5 Defining Non-static Member Classes

```
                         class ToplevelClass {                                   // (1)
                     private String headlines = "Shine the inner light";         // (2)
                   public NonStaticMemberClass makeInstance() {                  // (3)
                     return new NonStaticMemberClass();                     // (4)    }
                  public class NonStaticMemberClass {                       // (5) NSMC

                //  static int sf = 2003;                              // (6) Not OK.
           final static int fsf = 2003;                          // (7) OK.
            private String banner;                                // (8)
            public NonStaticMemberClass() { banner = headlines; }   // (9)
          public void print(String prefix) {
              System.out.println(prefix + banner + " in " + fsf);}// (10)     }
             }

public class Client {                                  // (11)
 public static void main(String[] args) {                   // (12)
 ToplevelClass topRef = new ToplevelClass();             // (13)
ToplevelClass.NonStaticMemberClass innerRef1 =
 topRef.makeInstance();                    // (14)
innerRef1.print("innerRef1: ");                       // (15)
//  ToplevelClass.NonStaticMemberClass innerRef2 =     //
 new ToplevelClass.NonStaticMemberClass(); // (16) Not OK.
ToplevelClass.NonStaticMemberClass innerRef3 =
 topRef.new NonStaticMemberClass();         // (17)
 innerRef3.print("innerRef3: ");                        // (18)     }
}
```

Output from the program:

```
innerRef1: Shine the inner light in 2003
innerRef3: Shine the inner light in 2003
```

A special form of the new operator is used to instantiate a non-static member class.

<enclosing object reference>.new <non-static member class constructor call>

The <enclosing object reference> in the object creation expression evaluates to an instance of the enclosing class in which the designated non-static member class is defined. A new instance of the non-static member class is created and associated with the indicated instance of the enclosing class. Note that the expression returns a reference value that denotes a new instance of the non-static member class. It is illegal to specify the full name of the non-static member class in the constructor call, as the enclosing context is already given by the <enclosing object reference>.

The non-static method makeInstance() at (3) in the class ToplevelClass creates an instance of the NonStaticMemberClass using the new operator, as shown at (4):

```
return new NonStaticMemberClass();                    // (4)
```

This creates an instance of a non-static member class in the context of the instance of the enclosing class on which the makeInstance() method is invoked. The new operator in the statement at (4) has an implicit this reference as the <enclosing object reference>, since the non-static member class is directly defined in the context of the object denoted by the this reference:

```
return this.new NonStaticMemberClass();               // (4')
```

The makeInstance() method is called at (14). This associates a new object of the NonStaticMemberClass with the object denoted by the reference topRef. This object of the NonStaticMemberClass is denoted by the reference innerRef1. The reference innerRef1 can then be used in the normal way to access members of this object of the non-static member class as shown at (15).

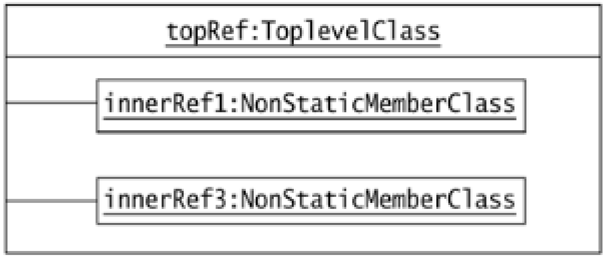
An attempt to create an instance of the non-static member class without an outer instance, using the new operator with the full name of the inner class, as shown at (16), results in a compile-time error.

The special form of the new operator is also used in the object creation expression at (17).

```
ToplevelClass.NonStaticMemberClass innerRef3 =
topRef.new NonStaticMemberClass();        // (17)
```

The reference topRef denotes an object of the class ToplevelClass. After the execution of the statement at (17), the ToplevelClass object has two instances of the non-static member class NonStaticInnerClass associated with it. This is depicted in Figure 7.2, where the outer object (denoted by topRef) of class ToplevelClass is shown with its two associated inner objects (denoted by references innerRef1 and innerRef3, respectively) right after the execution of the statement at (17). In other words, multiple objects of the non-static member classes can be associated with an object of an enclosing class at runtime.

Figure 7.2. Outer Object with Associated Inner Objects


topRef:ToplevelClass
innerRef1:NonStaticMemberClass
innerRef3:NonStaticMemberClass

Accessing Members in Enclosing Context

An implicit reference to the enclosing object is always available in every method and constructor of a non-static member class. A method can explicitly use this reference with a special form of the this construct, as explained in the next example.

From within a non-static member class, it is possible to refer to all members in the enclosing class directly. An example is shown at (9), where the field headlines from the enclosing class is accessed in the non-static member class. It is also possible to explicitly refer to members in the enclosing class, but this requires special usage of the this reference. One might be tempted to define the constructor at (9) as follows:

```
public NonStaticMemberClass() { this.banner = this.headlines; }
```

The reference this.banner is correct, because the field banner certainly belongs to the current object (denoted by this) of NonStaticMemberClass, but this.headlines cannot possibly work, as the current object (indicated by this) of NonStaticMember Class has no field headlines. The correct syntax is the following:

```
public NonStaticMemberClass() { this.banner = ToplevelClass.this.headlines; }
```

The expression

```
<enclosing class name>.this
```

evaluates to a reference that denotes the enclosing object (of class <enclosing class name>) of the current instance of a non-static member class.

Accessing Hidden Members

Fields and methods in the enclosing context can be hidden by fields and methods with the same names in the non-static member class. The special form of the this syntax can be used to access members in the enclosing context, somewhat analogous to using the keyword super in subclasses to access hidden superclass members.

Example 7.6 Special Form of this and new Constructs in Non-static Member Classes

```
// Filename: Client2.java
class TLClass {                                          // (1) TLC
private String id = "TLClass object ";                   // (2)
public TLClass(String objId) { id = id + objId; }        // (3)
 public void printId() {                                 // (4)
System.out.println(id);
   }
```

```
    class InnerB {                                    // (5) NSMC
  private String id = "InnerB object ";                 // (6)
  public InnerB(String objId) { id = id + objId; }      // (7)
   public void printId() {                            // (8)
   System.out.print(TLClass.this.id + " : ");         // (9)
  System.out.println(id);                           // (10)        }

        class InnerC {                                // (11) NSMC
  private String id = "InnerC object ";             // (12)
  public InnerC(String objId) { id = id + objId; }  // (13)
  public void printId() {                         // (14)
   System.out.print(TLClass.this.id + " : ");    // (15)
    System.out.print(InnerB.this.id + " : ");    // (16)
  System.out.println(id);                      // (17)            }
            public void printIndividualIds() {               // (18)
  TLClass.this.printId();                      // (19)
   InnerB.this.printId();                      // (20)               printId();                                  // (21)

            }
        }
    }
  }
  public class OuterInstances {                                  // (22)
   public static void main(String[] args) {                      // (23)
  TLClass a = new TLClass("a");                               // (24)
  TLClass.InnerB b = a.new InnerB("b");                          // (25)
     b.printId();                                    // (26)
   TLClass.InnerB.InnerC c = b.new InnerC("c");                 // (27)
     c.printId();                                    // (28)
  TLClass.InnerB.InnerC d = b.new InnerC("d");                  // (29)
   d.printId();                                     // (30)
  TLClass.InnerB bb = new TLClass("aa").new InnerB("bb");     // (31)
   bb.printId();                                    // (32)
   TLClass.InnerB.InnerC cc = bb.new InnerC("cc");            // (33)
   cc.printId();                                    // (34)
   TLClass.InnerB.InnerC ccc =
            new TLClass("aaa").new InnerB("bbb").new InnerC("ccc"); // (35)

    ccc.printId();                                   // (36)

    System.out.println("------------");
```

```
        ccc.printIndividualIds();                          // (37)     }
}
```

Output from the program:

```
TLClass object a : InnerB object b
TLClass object a : InnerB object b : InnerC object c
TLClass object a : InnerB object b : InnerC object d
TLClass object aa : InnerB object bb
TLClass object aa : InnerB object bb : InnerC object cc
TLClass object aaa : InnerB object bbb : InnerC object ccc
------------
TLClass object aaa
TLClass object aaa : InnerB object bbb
TLClass object aaa : InnerB object bbb : InnerC object ccc
```

Example 7.6 illustrates the special form of the this construct employed to access members in the enclosing context, and also demonstrates the special form of the new construct employed to create instances of non-static member classes. The example shows the non-static member class InnerC at (11), which is nested in the non-static member class InnerB at (5), which in turn is nested in the top-level class TLClass at (1). All three classes have a private non-static String field named id and a non-static method named printId. The member name in the nested class hides the name in the enclosing context. These members are not overridden in the nested classes, as no inheritance is involved. In order to refer to the hidden members, the nested class can use the special this construct, as shown at (9), (15), (16), (19), and (20). Within the nested class InnerC, the three forms used in the following statements to access its field id are equivalent:

```
System.out.println(id);                    // (17)
System.out.println(this.id);               // (17a)
System.out.println(InnerC.this.id);        // (17b)
```

The main() method at (23) uses the special syntax of the new operator to create objects of non-static member classes and associate them with enclosing objects. An instance of class InnerC (denoted by c) is created at (27) in the context of an instance of class InnerB (denoted by b), which was created at (25) in the context of an instance of class TLClassA (denoted by a), which in turn was created at (24). The reference c is used at (28) to invoke the method printId() declared at (14) in the nested class InnerC. This method prints the field id from all the objects associated with an instance of the nested class InnerC.

When the intervening references to an instance of a non-static member class are of no interest (i.e., if the reference values need not be stored in variables), then the new operator can be chained as shown at (31) and (35).

Note that the (outer) objects associated with the instances denoted by the references c, cc, and ccc are distinct, as evident from the program output. However, the instances denoted by references c and d have the same outer objects associated with them.

Inheritance Hierarchy and Enclosing Context

Inner classes can extend other classes, and vice versa. An inherited field (or method) in an inner subclass can hide a field (or method) with the same name in the enclosing context. Using the simple name to access this member will access the inherited member, not the one in the enclosing context.

Example 7.7 illustrates the situation outlined earlier. The standard form of the this reference is used to access the inherited member as shown at (4). The keyword super would be another alternative. To access the member from the enclosing context, the special form of the this reference together with the enclosing class name is used as shown at (5).

Example 7.7 Inheritance Hierarchy and Enclosing Context

```
class Superclass {
   protected double x = 3.0e+8;
}

class TopLevelClass {                    // (1) Top-level Class
private double x = 3.14;

   class Inner extends Superclass {       // (2) Non-static member Class
  public void printHidden() {        // (3)

         // (4) x from superclass:
         System.out.println("this.x: " + this.x);
         // (5) x from enclosing context:
         System.out.println("TopLevelClass.this.x: "+TopLevelClass.this.x);        }
      }

}


public class HiddenAndInheritedAccess {

   public static void main(String[] args) {
      TopLevelClass.Inner ref = new TopLevelClass().new Inner();
 ref.printHidden();
   }
}
```

Output from the program:

```
this.x: 3.0E8
TopLevelClass.this.x: 3.14
```

Some caution should be exercised when extending an inner class. Some of the subtleties involved are illustrated by Example 7.8.

The non-static member class InnerA, declared at (2) in class OuterA, is extended by the subclass SomeUnrelatedClass at (3). Note that SomeUnrelatedClass and class OuterA are not related in any way. An instance of subclass SomeUnrelatedClass is created at (8). An instance of class OuterA is explicitly passed as argument in the constructor call to SomeUnrelatedClass. The constructor at (4) for SomeUnrelatedClass has a special super() call in its body at (5). This call ensures that the constructor of the superclass InnerA has an outer object (denoted by the reference outerRef) to bind to. Using the standard super() call in the subclass constructor is not adequate, because it does not provide an outer instance for the superclass constructor to bind to. The non-default constructor at (4) and the outerRef.super() expression at (5) are mandatory to set up the proper relationships between the objects involved.

The outer object problem mentioned above does not arise if the subclass that extends an inner class is also declared within an outer class that extends the outer class of the superclass. This situation is illustrated at (6) and (7): classes InnerB and OuterB extend classes InnerA and OuterA, respectively. The type InnerA is inherited by class OuterB from its superclass OuterA. Thus an object of class OuterB can act as an outer object for an instance of class InnerA. The object creation expression at (9)

```
new OuterB().new InnerB();
```

creates an OuterB object and implicitly passes its reference to the default constructor of class InnerB. The default constructor of class InnerB invokes the default constructor of its superclass InnerA by calling super() and passing it the reference of the OuterB object, which the superclass constructor can readily bind to.

Example 7.8 Extending Inner Classes

```
class OuterA {                              // (1)
   class InnerA { }                         // (2)
}
class SomeUnrelatedClass extends OuterA.InnerA { // (3) Extends NSMC at (2)
  // (4) Mandatory non-default constructor

   SomeUnrelatedClass(OuterA outerRef) {
  outerRef.super();                    // (5) Explicit super() call     }
}

class OuterB extends OuterA {                  // (6) Extends class at (1)
class InnerB extends OuterB.InnerA { }       // (7) Extends NSMC at (2)
}

public class Extending {
   public static void main(String[] args) {

       // (8) Outer instance passed explicitly in constructor call:
 new SomeUnrelatedClass(new OuterA());
       // (9) No outer instance passed explicitly in constructor call to InnerB:
   new OuterB().new InnerB();
   }
}
```

7.4 Local Classes

A local class is an inner class that is defined in a block. This could be a method body, a constructor, a local block, a static initializer, or an instance initializer.

Blocks in a non-static context have a this reference available, which denotes an instance of the class containing the block. An instance of a local class, which is declared in such a non-static block, has an instance of the enclosing class associated with it. This gives such a non-static local class much of the same capability as a non-static member class.

However, if the block containing a local class declaration is defined in a static context (i.e., a static method or a static initializer), then the local class is implicitly static in the sense that its instantiation does not require any outer object. This aspect of local classes is reminiscent of static member classes. However, note that a local class cannot be specified with the keyword static.

Some restrictions that apply to local classes are

· Local classes cannot have static members, as they cannot provide class-specific services. However, final static fields are allowed, as these are constants. This is illustrated in Example 7.9 at (1) and (2) in the NonStaticLocal class, and also by the StaticLocal class at (11) and (12).

· Local classes cannot have any accessibility modifier. The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations.

Example 7.9 Access in Enclosing Context

```
class Base {
    protected int nsf1;
}
class TLCWithLocalClasses {            // Top level Class
private double nsf1;               // Non-static field
private int    nsf2;               // Non-static field
 private static int sf;             // Static field

    void nonStaticMethod(final int fp) { // Non-static Method
final int flv  = 10;           // final local variable
  final int hlv  = 30;           // final (hidden) local variable
   int nflv = 20;           // non-final local variable
        class NonStaticLocal extends Base { // Non-static local class
  //        static int f1;       // (1) Not OK. Static members not allowed.
 final static int f2 = 10;  // (2) final static members allowed.

   int    f3  = fp;    // (3) final param from enclosing method.

 int    f4  = flv;   // (4) final local var from enclosing method.         //  double

 f5  = nflv;  // (5) Not OK. Only finals from enclosing method.

 double f6  = nsf1;         // (6) Inherited from superclass.

 double f6a = this.nsf1;    // (6a) Inherited from superclass.

   double f6b = super.nsf1;   // (6b) Inherited from superclass.

   double f7  = TLCWithLocalClasses.this.nsf1;// (7) In enclosing object.

     int    f8  = nsf2;         // (8)  In enclosing object.

  int    f9  = sf;           // (9)  static from enclosing class.

     int    hlv;                // (10) Hides local variable.        }

    }
    static void staticMethod(final int fp) { // Static Method
 final int flv  = 10;           // final local variable
 final int hlv  = 30;           // final (hidden) local variable
int nflv = 20;           // non-final local variable
        class StaticLocal extends Base { // Static local class
//        static int f1;       // (11) Not OK. Static members not allowed.
   final static int f2 = 10;  // (12) final static members allowed.
  int    f3  = fp;    // (13) final param from enclosing method.
  int    f4  = flv;   // (14) final local var from enclosing method.
  //  double f5  = nflv;  // (15) Not OK. Only finals from enclosing method.
     double f6  = nsf1;         // (16) Inherited from superclass.
 double f6a = this.nsf1;    // (16a) Inherited from superclass.
double f6b = super.nsf1;   // (16a) Inherited from superclass.
   //  double f7  = TLCWithLocalClasses.this.nsf1;//(17)No enclosing object.
 //  int    f8  = nsf2;         // (18)  No enclosing object.
  int    f9  = sf;           // (19)  static from enclosing class.
  int    hlv;                // (20) Hides local variable.        }
    }
}
```

Accessing Declarations in Enclosing Context

Example 7.9 illustrates how a local class can access declarations in its enclosing context. Declaring a local class in a static or a non-static block, influences what the class can access in the enclosing context.

Accessing Local Declarations in the Enclosing Block

A local class can access final local variables, final method parameters, and final catch-block parameters in the scope of the local context. Such final variables are also read-only in the local class. This situation is shown at (3) and (4), where the final parameter fp and the final local variable flv of the method nonStaticMethod() in the NonStaticLocal class are accessed. This also applies to static local classes, as shown at (13) and (14) in the StaticLocal class.

Access to non-final local variables is not permitted from local classes, as shown at (5) and (15).

Declarations in the enclosing block of a local class can be hidden by declarations in the local class. At (10) and (20), the field hlv hides the local variable by the same name in the enclosing method. There is no way for the local class to refer to such hidden declarations.

Accessing Members in the Enclosing Class

A local class can access members inherited from its superclass in the usual way. The field nsf1 in the superclass Base is inherited by the local subclass NonStaticLocal. This inherited field is accessed in the NonStaticLocal class as shown at (6), (6a), and (6b) by using the field's simple name, the standard this reference, and the super keyword, respectively. This also applies for static local classes as shown at (16), (16a), and (16b).

Fields and methods in the enclosing class can be hidden by member declarations in the local class. The non-static field nsf1, inherited by the local classes, hides the field by the same name in the TLCWithLocalClasses class. The special form of the this construct can be used in non-static local classes for explicit referencing of members in the enclosing class, regardless of whether these members are hidden or not.

double f7 = TLCWithLocalClasses.this.nsf1; // (7)

However, the special form of the this construct cannot be used in a static local class, as shown at (17), since it does not have any notion of an outer object. The static local class cannot refer to such hidden declarations.

A non-static local class can access both static and non-static members defined in the enclosing class. The non-static field nsf2 and static field sf are defined in the enclosing TLCWithLocalClasses class. They are accessed in the NonStaticLocal class at (8) and (9), respectively. The special form of the this construct can also be used in non-static local classes, as previously mentioned.

However, a static local class can only directly access members defined in the enclosing class that are static. The static field sf in the TLCWithLocalClasses class is accessed in the StaticLocal class at (19), but the non-static field nsf1 cannot be accessed, as shown at (18).

Instantiating Local Classes

Clients outside the scope of a local class cannot instantiate the class directly because such classes are, after all, local. A local class can be instantiated in the block in which it is defined. Like a local variable, a local class must be declared before being used in the block.

A method can return instances of any local class it declares. The local class type must then be assignable to the return type of the method. The return type cannot be the same as the local class type, since this type is not accessible outside of the method. A supertype of the local class must be specified as the return type. This also means that, in order for the objects of the local class to be useful outside the method, a local class should implement an interface or override the behavior of its supertypes.

Example 7.10 illustrates how clients can instantiate local classes. The non-static local class Circle at (5) is defined in the non-static method createCircle() at (4), which has the return type Shape. The static local class Map at (8) is defined in the static method createMap() at (7), which has the return type IDrawable. The inheritance hierarchy of the local classes and their supertypes Shape and IDrawable is depicted in Figure 6.5. The main() method creates a polymorphic array drawables of type IDrawable[] at (10), which is initialized at lines (10) through (13) with instances of the local classes.

Example 7.10 Instantiating Local Classes

```
interface IDrawable {                    // (1)
   void draw();
}
class Shape implements IDrawable {      // (2)
public void draw() { System.out.println("Drawing a Shape."); }
}

class Painter {                         // (3) Top-level Class
 public Shape createCircle(final double radius) { // (4) Non-static Method
 class Circle extends Shape {    // (5) Non-static local class
  public void draw() {
               System.out.println("Drawing a Circle of radius: " + radius);             }
      }
      return new Circle();           // (6) Object of non-static local class
 }
   public static IDrawable createMap() {  // (7) Static Method

 class Map implements IDrawable {   // (8) Static local class

  public void draw() { System.out.println("Drawing a Map."); }

 }

      return new Map();                 // (9) Object of static local class

  }

}
```

```
     public class LocalClassClient {
   public static void main(String[] args) {
      IDrawable[] drawables = {       // (10)
 new Painter().createCircle(5),// (11) Object of non-static local class
Painter.createMap(),          // (12) Object of static local class
new Painter().createMap()     // (13) Object of static local class
 };
      for (int i = 0; i < drawables.length; i++)     // (14)
drawables[i].draw();

      System.out.println("Local Class Names:");
System.out.println(drawables[0].getClass());   // (15)
System.out.println(drawables[1].getClass());   // (16)     }
}
```

Output from the program:

```
Drawing a Circle of radius: 5.0
Drawing a Map.
Drawing a Map.
Local Class Names:
class Painter$1$Circle
class Painter$1$Map
```

Creating an instance of a non-static local class requires an instance of the enclosing class. The non-static method createCircle() is invoked on the instance of the enclosing class to create an instance of the non-static local class, as shown at (11). In the non-static method, the reference to the instance of the enclosing context is passed implicitly in the constructor call of the non-static local class at (6).

A static method can be invoked either through the class name or through a reference of the class type. An instance of a static local class can be created either way by calling the createMap() method as shown at (12) and (13). As might be expected, no outer object is involved.

As references to a local class cannot be declared outside of the local context, the functionality of the class is only available through supertype references. The method draw() is invoked on objects in the array at (14). The program output indicates which objects were created. In particular, note that the final parameter radius of the method createCircle() at (4) is accessed by the draw() method of the local class Circle at (5). An instance of the local class Circle is created at (11) by a call to the method createCircle(). The draw() method is invoked on this instance of the local class Circle in the loop at (14). The value of the final parameter radius is still accessible to the draw() method invoked on this instance, although the call to the method createCircle(), which created the instance in the first place, has completed. Values of final local variables continue to be available to instances of local classes whenever these values are needed.

The output in Example 7.10 also shows the actual names of the local classes. In fact, the local class names are reflected in the class filenames.

Another use of local classes is shown in the following example. The code shows how local classes can be used, together with assertions, to implement certain kinds of postconditions. The basic idea is that a computation wants to save some data that is later required when checking a postconditon. For example, a deposit is made into an account, and we want to check that the transaction is valid after it is done. The computation can save the old balance before the transaction, so that the new balance can be correlated with the old balance after the transaction.

The local class Auditor at (2) acts as a repository for data that needs to be retrieved later to check the postconditon. Note that it accesses the final parameter, but declarations that follow its declaration would not be accessible. The assertion in the method check() at (4) ensures that the postcondition is checked, utilizing the data that was saved when the Auditor object was constructed at (5).

```
class Account {
    int balance;

    /** (1) Method makes a deposit into an account. */
void deposit(final int amount) {

        /** (2) Local class to save the necessary data and to check
  that the transaction was valid. */
        class Auditor {

            /** (3) Stores the old balance. */
            private int balanceAtStartOfTransaction = balance;
            /** (4) Checks the postcondition. */             void check() {
                assert balance - balanceAtStartOfTransaction == amount;             }
        }

        Auditor auditor = new Auditor(); // (5) Save the data.
   balance += amount;                // (6) Do the transaction.
auditor.check();                 // (7) Check the postcondition.
  }
   public static void main(String[] args) {
        Account ac = new Account();
        ac.deposit(250);
    }

}
```

7.5 Anonymous Classes

Classes are usually first defined and then instantiated using the new operator. Anonymous classes combine the process of definition and instantiation into a single step. Anonymous classes are defined at the location they are instantiated, using additional syntax with the new operator. As these classes do not have a name, an instance of the class can only be created together with the definition.

An anonymous class can be defined and instantiated in contexts where a reference can be used (i.e., as expressions that evaluate to a reference denoting an object). Anonymous classes are typically used for creating objects on the fly in contexts such as the value in a return statement, an argument in a method call, or in initialization of variables. Anonymous classes are heavily used to implement event listeners in GUI-based applications.

Like local classes, anonymous classes can be defined in static or non-static context. The keyword static is never used.

Extending an Existing Class

The following syntax can be used for defining and instantiating an anonymous class that extends an existing class specified by <superclass name>: new <superclass name> (<optional argument list>) { <member declarations> } Optional arguments can be specified, which are passed to the superclass constructor. Thus, the superclass must provide a constructor corresponding to the arguments passed. No extends clause is used in the construct. Since an anonymous class cannot define constructors (as it does not have a name), an instance initializer can be used to achieve the same effect as a constructor. Only non-static members and final static fields can be declared in the class body.

Example 7.11 Defining Anonymous Classes

```
interface IDrawable {                    // (1)
void draw();
}
class Shape implements IDrawable {            // (2)
public void draw() { System.out.println("Drawing a Shape."); }
 }

class Painter {                              // (3) Top-level Class
   public Shape createShape() {               // (4) Non-static Method
 return new Shape(){                   // (5) Extends superclass at (2)
 public void draw() { System.out.println("Drawing a new Shape."); }
  };
   }
   public static IDrawable createIDrawable() { // (7) Static Method
 return new IDrawable(){                // (8) Implements interface at (1)
 public void draw() {
 System.out.println("Drawing a new IDrawable.");


            }

        };

    }

}



public class AnonClassClient {

    public static void main(String[] args) { // (9)

IDrawable[] drawables = {             // (10)

new Painter().createShape(),     // (11) non-static anonymous class
Painter.createIDrawable(),       // (12) static anonymous class

  new Painter().createIDrawable()  // (13) static anonymous class

};

        for (int i = 0; i < drawables.length; i++)     // (14)

  drawables[i].draw();

   System.out.println("Anonymous                          Class                          Names:");
System.out.println(drawables[0].getClass());                                    //          (15)
System.out.println(drawables[1].getClass());   // (16)     }

}
```

Output from the program:

```
Drawing a new Shape.
Drawing a new IDrawable.
Drawing a new IDrawable.
Anonymous Class Names:
class Painter$1

class Painter$2
```

Class definitions from Example 7.11, an adaptation of Example 7.10 to anonymous classes, are shown below. The non-static method createShape() at (4) defines a non-static anonymous class at (5), which extends the superclass Shape. The anonymous class overrides the inherited method draw(). As references to an anonymous class cannot be declared, the functionality of the class is only available through superclass references. Usually it makes sense to override methods from the superclass. Any other members declared in an anonymous class cannot be accessed by an external client.

```
// ...

class Shape implements IDrawable {          // (2)

  public void draw() { System.out.println("Drawing a Shape."); }

 }

class Painter {                            // (3) Top-level Class

public Shape createShape() {            // (4) Non-static Method

  return new Shape() {               // (5) Extends superclass at (2)

 public void draw() { System.out.println("Drawing a new Shape."); }

   };

    }

   // ...

}

// ...
```

Implementing an Interface

The following syntax can be used for defining and instantiating an anonymous class that implements an interface specified by <interface name>:

```
 new <interface name>() { <member declarations> }
```

An anonymous class provides a single interface implementation, and no arguments are passed. The anonymous class implicitly extends the Object class. Note that no implements clause is used in the construct. The class body has the same restrictions as previously noted for anonymous classes extending an existing class.

An anonymous class implementing an interface is shown below. Details can be found in Example 7.11. The static method createIDrawable() at (7) defines a static anonymous class at (8), which implements the interface IDrawable, by providing an implementation of the method draw(). The functionality of objects of an anonymous class that implements an interface is available through references of the interface type and the Object type, (i.e., the supertypes).

```
interface IDrawable {                          // (1) Interface
 void draw();
}
// ...
class Painter {                               // (3) Top-level Class
 // ...
   public static IDrawable createIDrawable() { // (7) Static Method
return new IDrawable(){                 // (8) Implements interface at (1)
 public void draw() {
              System.out.println("Drawing a new IDrawable.");
  }
       };
   }
} // ...
```

The following code is an example of a typical use of anonymous classes in building GUI-application. The anonymous class at (1) implements the ActionListener interface that has the method actionPerformed(). When the addActionListener() method is called on the GUI-button denoted by the reference quitButton, the anonymous class is instantiated and the object reference value passed as a parameter to the method. The method addActionListener() of the GUI-button can use the reference value to invoke the method actionPerformed() in the ActionListener object.

```
quitButton.addActionListener(
       new ActionListener() {    // (1) Anonymous class implements interface.

  // Invoked when the user clicks the quit button.

public void actionPerformed(ActionEvent evt) {

  System.exit(0);   // (2) Terminates the program.

    }

       }

);
```

Instantiating Anonymous Classes

The discussion on instantiating local classes is also valid for instantiating anonymous classes. The class AnonClassClient in Example 7.11 creates one instance at (11) of the non-static anonymous class defined at (5), and two instances at (12) and (13) of the static anonymous class defined at (8). The program output shows the polymorphic behavior and the runtime types of the objects. Similar to a non-static local class, an instance of a non-static anonymous class has an instance of its enclosing class at (11). An enclosing instance is not mandatory for creating objects of a static anonymous class, as shown at (12).

The names of the anonymous classes at runtime are also shown in the program output in Example 7.11. They are also the names used to designate their respective class files. Anonymous classes are not so anonymous after all.

Accessing Declarations in Enclosing Context

Access rules for local classes also apply to anonymous classes. Example 7.12 is an adaptation of Example 7.9 and illustrates the access rules for anonymous classes. The local classes in Example 7.9 have been adapted to anonymous classes in Example 7.12. The TLCWithAnonClasses class has two methods: one non-static and the other static, which return an instance of a non-static and a static anonymous class, respectively. Both anonymous classes extend the Base class.

Anonymous classes can access final variables only in the enclosing context. Inside the definition of a non-static anonymous class, members of the enclosing context can be referenced using the <enclosing class name>.this construct. Non-static anonymous classes can also access any non-hidden members by their simple names in the enclosing context, whereas static anonymous classes can only access non-hidden static members.

Example 7.12 Accessing Declarations in Enclosing Context

```
class Base {
   protected int nsf1;
}

class TLCWithAnonClasses {               // Top level Class
  private double nsf1;                 // Non-static field
private int    nsf2;                 // Non-static field
private static int sf;               // Static field
   Base nonStaticMethod(final int fp) { // Non-static Method
 final int flv  = 10;             // final local variable
  final int hlv  = 30;            // final (hidden) local variable
  int nflv = 20;            // non-final local variable
  return new Base() {           // Non-static anonymous class
 //       static int f1;      // (1) Not OK. Static members not allowed.
```

```
final static int f2 = 10; // (2) final static members allowed.
   int    f3  = fp;    // (3) final param from enclosing method.
 int    f4  = flv;   // (4) final local var from enclosing method.
//  double f5  = nflv;  // (5) Not OK. Only finals from enclosing method.


       double f6  = nsf1;        // (6) Inherited from superclass.
   double f6a = this.nsf1;   // (6a) Inherited from superclass.
 double f6b = super.nsf1;  // (6b) Inherited from superclass.
 double f7  = TLCWithAnonClasses.this.nsf1; // (7) In enclosing object.
   int    f8  = nsf2;        // (8)  In enclosing object.
 int    f9  = sf;          // (9)  static from enclosing class.
 int    hlv;               // (10) Hides local variable.
 };
   }


   static Base staticMethod(final int fp) { // Static Method
 final int flv  = 10;                 // final local variable
 final int hlv  = 30;                 // final (hidden) local variable
  int nflv = 20;                 // non-final local variable
     return new Base() {           // Static anonymous class
  //        static int f1;      // (11) Not OK. Static members not allowed.
   final static int f2 = 10; // (12) final static members allowed.
  int    f3  = fp;    // (13) final param from enclosing method.
int    f4  = flv;   // (14) final local var from enclosing method.
 //  double f5  = nflv;  // (15) Not OK. Only finals from enclosing method.
  double f6  = nsf1;        // (16 ) Inherited from superclass.
 double f6a = this.nsf1;   // (16a) Inherited from superclass.
 double f6b = super.nsf1;  // (16b) Inherited from superclass.
   //  double f7  = TLCWithAnonClasses.this.nsf1; //(17) No enclosing object.
       //  int    f8  = nsf2;        // (18)  No enclosing object.
  int    f9  = sf;          // (19)  static from enclosing class.
     int    hlv;                // (20) Hides local variable.
   };
    }
}
```

08

Multi-threading

8.1 Multi-tasking

Multitasking allows several activities to occur concurrently on the computer. A distinction is usually made between:

- Process-based multitasking
- Thread-based multitasking

At the coarse-grain level there is process-based multitasking, which allows processes (i.e., programs) to run concurrently on the computer. A familiar example is running the spreadsheet program while also working with the word-processor. At the fine-grain level there is thread-based multitasking, which allows parts of the same program to run concurrently on the computer. A familiar example is a word-processor that is printing and formatting text at the same time. This is only feasible if the two tasks are performed by two independent paths of execution at runtime. The two tasks would correspond to executing parts of the program concurrently. The sequence of code executed for each task defines a separate path of execution, and is called a thread (of execution).

In a single-threaded environment only one task at a time can be performed. CPU cycles are wasted, for example, when waiting for user input. Multitasking allows idle CPU time to be put to good use.

Some advantages of thread-based multitasking as compared to process-based multitasking are

- threads share the same address space
- context switching between threads is usually less expensive than between processes
- cost of communication between threads is relatively low

Java supports thread-based multitasking and provides high-level facilities for multithreaded programming. Thread safety is the term used to describe the design of classes that ensure that the state of their objects is always consistent, even when the objects are used concurrently by multiple threads.

8.2 Overview of Threads

A thread is an independent sequential path of execution within a program. Many threads can run concurrently within a program. At runtime, threads in a program exist in a common memory space and can, therefore, share both data and code, that is, they are lightweight compared to processes. They also share the process running the program.

Every thread in Java is created and controlled by a unique object of the java.lang.Thread class. Often the thread and its associated Thread object are thought of as being synonymous.

Threads make the runtime environment asynchronous, allowing different tasks to be performed concurrently. Using this powerful paradigm in Java centers around understanding the following aspects of multithreaded programming:

· creating threads and providing the code that gets executed by a thread.
· accessing common data and code through synchronization
· transitioning between thread states.

The Main Thread

The runtime environment distinguishes between user threads and daemon threads. As long as a user thread is alive, the JVM does not terminate. A daemon thread is at the mercy of the runtime system: it is stopped if there are no more user threads running, thus terminating the program. Daemon threads exist only to serve user threads.

When a standalone application is run, a user thread is automatically created to execute the main() method. This thread is called the main thread. If no other user threads are spawned, the program terminates when the main() method finishes executing. All other threads, called child threads, are spawned from the main thread, inheriting its user-thread status. The main() method can then finish, but the program will keep running until all the user threads have finished. Calling the setDaemon(boolean) method in the Thread class marks the status of the thread as either daemon or user, but this must be done before the thread is started. Any attempt to change the status after the thread has been started, throws an IllegalThreadStateException. Marking all spawned threads as daemon threads ensures that the application terminates when the main thread dies.

When a GUI application is started, a special thread is automatically created to monitor the user–GUI interaction. This user thread keeps the program running, allowing interaction between the user and the GUI, even though the main thread might have died after the main() method finished executing.

8.3 Thread Creation

A thread in Java is represented by an object of the Thread class. Implementing threads is achieved in one of two ways:

- implementing the java.lang.Runnable interface
- extending the java.lang.Thread class

Implementing the Runnable Interface

The Runnable interface has the following specification, comprising one method prototype declaration:

```
 public interface Runnable {

    void run();

}
```

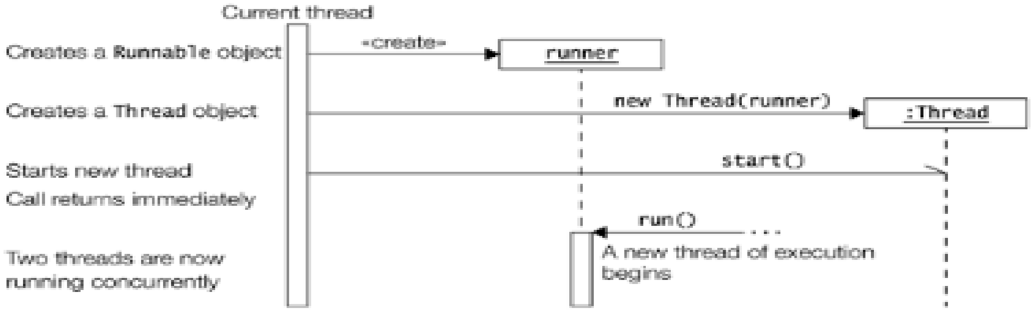
A thread, which is created based on an object that implements the Runnable interface, will execute the code defined in the public method run(). In other words, the code in the run() method defines an independent path of execution and thereby the entry and the exits for the thread. The thread ends when the run() method ends, either by normal completion or by throwing an uncaught exception.

The procedure for creating threads based on the Runnable interface is as follows:

1. A class implements the Runnable interface, providing the run() method that will be executed by the thread. An object of this class is a Runnable object.
2. An object of Thread class is created by passing a Runnable object as argument to the Thread constructor. The Thread object now has a Runnable object that implements the run() method.
3. The start() method is invoked on the Thread object created in the previous step. The start() method returns immediately after a thread has been spawned.

The run() method of the Runnable object is eventually executed by the thread represented by the Thread object on which the start() method was invoked. This sequence of events is illustrated in Figure 8.1.

Figure 8.1. Spawning Threads Using a Runnable Object



The following is a summary of important constructors and methods from the java.lang.Thread class:

Thread(Runnable threadTarget)

Thread(Runnable threadTarget, String threadName)

The argument threadTarget is the object whose run() method will be executed when the thread is started. The argument threadName can be specified to give an explicit name for the thread, rather than an automatically generated one. A thread's name can be retrieved by using the getName() method.

static Thread currentThread()

This method returns a reference to the Thread object of the currently executing thread.

final String getName()
final void setName(String name)

The first method returns the name of the thread. The second one sets the thread's name to the argument.
void run()

The Thread class implements the Runnable interface by providing an implementation of the run() method. This implementation does nothing. Subclasses of the Thread class should override this method.

If the current thread is created using a separate Runnable object, then the Runnable object's run() method is called.

final void setDaemon(boolean flag)
final boolean isDaemon()

The first method sets the status of the thread either as a daemon thread or as a user thread, depending on whether the argument is true or false, respectively. The status should be set before the thread is started. The second method returns true if the thread is a daemon thread, otherwise, false.

void start()

This method spawns a new thread, that is, the new thread will begin execution as a child thread of the current thread. The spawning is done asynchronously as the call to this method returns immediately. It throws an IllegalThreadStateException if the thread was already started.

In Example 8.1, the class Counter implements the Runnable interface. The constructor for the Counter class ensures that each object of the Counter class will create a new thread by passing the Counter instance to the Thread constructor, as shown at (1). In addition, the thread is enabled for execution by the call to its start() method, as shown at (2). At (3), the class defines the run() method that constitutes the code executed by the thread. In each iteration the thread will sleep for 250 milliseconds after writing the current value of the counter, as shown at (4). While it is sleeping, other threads may run.

Example 8.1 Implementing the Runnable Interface

```
class Counter implements Runnable {

   private int currentValue;

   private Thread worker;

   public Counter(String threadName) {
      currentValue = 0;
                  worker = new Thread(this, threadName);   // (1) Create a new thread.

      System.out.println(worker);
      worker.start();                        // (2) Start the thread.
  }

   public int getValue() { return currentValue; }
   public void run() {                        // (3) Thread entry point
 try {
         while (currentValue < 5) {
            System.out.println(worker.getName() + ": " + (currentValue++));
Thread.sleep(250);              // (4) Current thread sleeps.
 }

      } catch (InterruptedException e) {

         System.out.println(worker.getName() + " interrupted.");
}
        System.out.println("Exit from thread: " + worker.getName());     }
}
 public class Client {
    public static void main(String[] args) {         Counter counterA = new Counter("Counter A");
 // (5) Create a thread.
        try {
            int val;
            do {
                val = counterA.getValue();       // (6) Access the counter value.
System.out.println("Counter value read by main thread: " + val);
   Thread.sleep(1000);              // (7) Current thread sleeps.
} while (val < 5);
        } catch (InterruptedException e) {
            System.out.println("main thread interrupted.");
    }
 System.out.println("Exit from main() method.");     }
}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Counter value read by main thread: 0
Counter A: 0
Counter A: 1
Counter A: 2
Counter A: 3
Counter value read by main thread: 4
Counter A: 4
Exit from thread: Counter A
Counter value read by main thread: 5
Exit from main() method.
```

The Client class uses the Counter class. It creates an object of class Counter at (5) and retrieves its value in a loop at (6). After each retrieval, it sleeps for 1,000 milliseconds at (7), allowing other threads to run.

Note that the main thread executing in the Client class sleeps for a longer time between iterations than the Counter thread, giving the Counter thread the opportunity to run as well. The Counter thread is a child thread of the main thread. It inherits the user-thread status from the main thread. If the code after statement at (5) in the main() method was removed, the main thread would finish executing before the child thread. However, the program would continue running until the child thread completed.

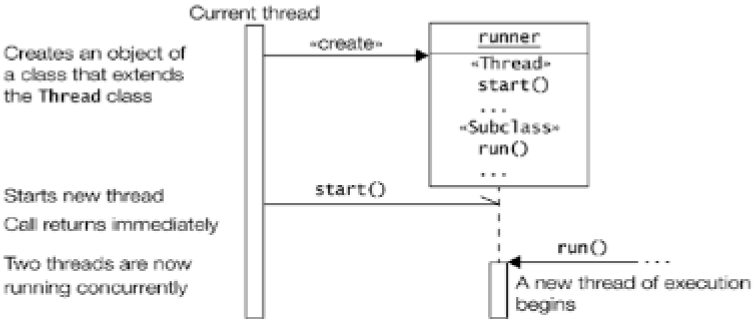
Since thread scheduling is not predictable and Example 8.1 does not enforce any synchronization between the two threads in accessing the counter value, the output shown may vary. The first line of the output shows the string representation of a Thread object: its name (Counter A), its priority (5), and its parent thread (main). The output from the main thread and the Counter A thread is interleaved. Not surprisingly, it also shows that the value in the Counter A thread was incremented faster than the main thread could access the counter's value after each increment.

Extending the Thread Class

A class can also extend the Thread class to create a thread. A typical procedure for doing this is as follows:

1. A class extending the Thread class overrides the run() method from the Thread class to define the code executed by the thread.
2. This subclass may call a Thread constructor explicitly in its constructors to initialize the thread, using the super() call.
3. The start() method inherited from the Thread class is invoked on the object of the class to make the thread eligible for running.

Figure 8.2. Spawning Threads‒Extending the Thread Class


Current thread
Creates an object of a class that extends the Thread class
«create»
runner
«Thread»
start()
...
«Subclass»
run()
...
Starts new thread
start()
Call returns immediately
Two threads are now running concurrently
run()
...
A new thread of execution begins

In Example 8.2, the Counter class from Example 8.1 has been modified to illustrate extending the Thread class. Note the call to the constructor of the superclass Thread at (1) and the invocation of the inherited start() method at (2) in the constructor of the Counter class. The program output shows that the Client class creates two threads and exits, but the program continues running until the child threads have completed. The two child threads are independent, each having its own counter and executing its own run() method.

The Thread class implements the Runnable interface, which means that this approach is not much different from implementing the Runnable interface directly. The only difference is that the roles of the Runnable object and the Thread object are combined in a single object.

The static method currentThread() in the Thread class can be used to obtain a reference to the Thread object associated with the current thread. An example of its usage is shown at (5) in Example 8.2:

```
Thread.currentThread().getName());   // (5) Current thread
```

Adding the following statement before the call to the start() method at (2) in Example 8.2:

```
setDaemon(true);
```

illustrates the daemon nature of threads. The program execution will now terminate after the main thread has completed, without waiting for the daemon Counter threads to finish normally.

Example 8.2 Extending the Thread Class

```
class Counter extends Thread {

    private int currentValue;

    public Counter(String threadName) {
        super(threadName);                      // (1) Initialize thread.
 currentValue = 0;
        System.out.println(this);
        start();                                // (2) Start this thread.
  }

    public int getValue() { return currentValue; }
public void run() {                             // (3) Override from superclass.
  try {
            while (currentValue < 5) {
                System.out.println(getName() + ": " + (currentValue++));
   Thread.sleep(250);              // (4) Current thread sleeps.
}
        } catch (InterruptedException e) {
            System.out.println(getName() + " interrupted.");
 }
        System.out.println("Exit from thread: " + getName());     }
}
```

```
public class Client {

   public static void main(String[] args) {

     System.out.println("Method main() runs in thread " +
Thread.currentThread().getName());   // (5) Current thread

      Counter counterA = new Counter("Counter A"); // (6) Create a thread.

  Counter counterB = new Counter("Counter B"); // (7) Create a thread.

      System.out.println("Exit from main() method.");     }

}
```

Possible output from the program:

```
Method main() runs in thread main
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Exit from main() method.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from thread: Counter A
Exit from thread: Counter B
```

When creating threads, there are two reasons why implementing the Runnable interface may be preferable to extending the Thread class:

· Extending the Thread class means that the subclass cannot extend any other class, whereas a class implementing the Runnable interface has this option.
· A class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

In Examples 8.1 and 8.2, the Thread object was created and the start() method called immediately to initiate the thread execution, as shown at (2). There are other ways of starting a thread. The call to the start() method can be factored out of the constructor, and the method can be called later, using a reference that denotes the thread object.

Inner classes are useful for implementing threads that do simple tasks. The anonymous class below will create a thread and start it:

```
(   new Thread() {
        public void run() {

            for(;;) System.out.println("Stop the world!");         }

    }).start();
```

8.4 Synchronization

Threads share the same memory space, that is, they can share resources. However, there are critical situations where it is desirable that only one thread at a time has access to a shared resource. For example, crediting and debiting a shared bank account concurrently amongst several users without proper discipline, will jeopardize the integrity of the account data. Java provides high-level concepts for synchronization in order to control access to shared resources.

Locks

A lock (a.k.a. monitor) is used to synchronize access to a shared resource. A lock can be associated with a shared resource. Threads gain access to a shared resource by first acquiring the lock associated with the resource. At any given time, at the most one thread can hold the lock (i.e., own the monitor) and thereby have access to the shared resource. A lock thus implements mutual exclusion (a.k.a. mutex).

In Java, all objects have a lock—including arrays. This means that the lock from any Java object can be used to implement mutual exclusion. By associating a shared resource with a Java object and its lock, the object can act as a guard, ensuring synchronized access to the resource. Only one thread at a time can access the shared resource guarded by the object lock.

The object lock mechanism enforces the following rules of synchronization:

· A thread must acquire the object lock associated with a shared resource, before it can enter the shared resource. The runtime system ensures that no other thread can enter a shared resource if another thread already holds the object lock associated with the shared resource. If a thread cannot immediately acquire the object lock, it is blocked, that is, it must wait for the lock to become available.

- When a thread exits a shared resource, the runtime system ensures that the object lock is also relinquished. If another thread is waiting for this object lock, it can proceed to acquire the lock in order to gain access to the shared resource.

Classes also have a class-specific lock that is analogous to the object lock. Such a lock is actually a lock on the java.lang.Class object associated with the class. Given a class A, the reference A.class denotes this unique Class object. The class lock can be used in much the same way as an object lock to implement mutual exclusion.

The keyword synchronized and the lock form the basis for implementing synchronized execution of code. There are two ways in which execution of code can be synchronized:

- synchronized methods
- synchronized blocks

Synchronized Methods

If the methods of an object should only be executed by one thread at a time, then the declaration of all such methods should be specified with the keyword synchronized. A thread wishing to execute a synchronized method must first obtain the object's lock (i.e., hold the lock) before it can enter the object to execute the method. This is simply achieved by calling the method. If the lock is already held by another thread, the calling thread waits. No particular action on the part of the program is necessary. A thread relinquishes the lock simply by returning from the synchronized method, allowing the next thread waiting for this lock to proceed.

Synchronized methods are useful in situations where methods can manipulate the state of an object in ways that can corrupt the state if executed concurrently. A stack implementation usually defines the two operations push and pop as synchronized, so that pushing and popping of elements are mutually exclusive operations. If several threads were to share a stack, then one thread would, for example, not be able to push an element on the stack while another thread was popping the stack. The integrity of the stack is maintained in the face of several threads accessing the state of the same stack. This situation is illustrated by Example 8.3.

In Example 8.3, the main() method in class Mutex creates a stack at (6), which is used by the two threads created at (7) and (8). The two threads continually push and pop the stack, respectively. The non-synchronized push() and pop() methods at (2a) and (4a) intentionally sleep at (3) and (5), respectively, between an update and the use of the value in the field topOfStack. This setup exaggerates the chances for the state of the stack being corrupted by one of the threads, while the other one is sleeping. The output from the program bears this out, when the methods are not declared synchronized. Non-synchronized updating of the value in the field topOfStack between the two threads is a disaster waiting to happen. This is an example of what is called a race condition. It occurs when two or more threads simultaneously update the same value, and as a consequence, leave the value in an undefined or inconsistent state.

From the output shown in Example 8.3, we can see that the main thread exits right after creating and starting the threads. The threads push and pop the stack. The stack state eventually gets corrupted, resulting in an ArrayOutOfBoundsException in the Pusher thread. The uncaught exception results in the demise of the Pusher thread, but the Popper thread continues.

Running the program in Example 8.3 with the synchronized version of the push() and pop() methods at (2b) and (4b), respectively, avoids the race condition. The method sleep() does not relinquish any lock that the thread might have on the current object. It is only relinquished when the synchronized method exits, guaranteeing mutually exclusive push-and-pop operations on the stack.

Example 8.3 Mutual Exclusion

```
class StackImpl {                              // (1)
 private Object[] stackArray;
  private int topOfStack;

  public StackImpl(int capacity) {


    stackArray = new Object[capacity];
    topOfStack = -1;
  }

  public boolean push(Object element) {            // (2a) non-synchronized

 //  public synchronized Boolean

 push(Object element) { // (2b) synchronized

if (isFull()) return false;

    ++topOfStack;

    try { Thread.sleep(1000); } catch (Exception ex) { }
 // (3) Sleep a little.
stackArray[topOfStack] = element;
    return true;
  }

  public Object pop() {                        // (4a) non-synchronized
//  public synchronized Object pop() {              // (4b) synchronized
  if (isEmpty()) return null;
        Object obj = stackArray[topOfStack];
stackArray[topOfStack] = null;
        try { Thread.sleep(1000); } catch (Exception ex) { } // (5) Sleep a little.
  topOfStack--;
        return obj;
    }
    public boolean isEmpty() { return topOfStack < 0; }
  public boolean isFull()  { return topOfStack >= stackArray.length - 1; } }

public class Mutex {
    public static void main(String[] args) {


        final StackImpl stack = new StackImpl(20);  // (6) Shared by the threads.
        (new Thread("Pusher") {                     // (7) Thread no. 1
   public void run() {
                for(;;) {
                    System.out.println("Pushed: " +
stack.push(new Integer(2003)));                 }
            }
        }).start();

        (new Thread("Popper") {                     // (8) Thread no. 2
    public void run() {
                for(;;) {
                    System.out.println("Popped: " + stack.pop());                 }
            }
        }).start();

        System.out.println("Exit from main().");     }
}
```

Possible output from the program:

```
Exit from main().
...
Pushed: true
Popped: 2003
Popped: 2003
```

```
Popped: null
...
Popped: null
java.lang.ArrayIndexOutOfBoundsException: -1        at StackImpl.push(Mutex.java:15)
        at Mutex$1.run(Mutex.java:41)
Popped: null
Popped: null
...
```

While a thread is inside a synchronized method of an object, all other threads that wish to execute this synchronized method or any other synchronized method of the object will have to wait. This restriction does not apply to the thread that already has the lock and is executing a synchronized method of the object. Such a method can invoke other synchronized methods of the object without being blocked. The non-synchronized methods of the object can of course be called at any time by any thread.

Not surprisingly, static methods synchronize on the class lock. Acquiring and relinquishing a class lock by a thread in order to execute a static synchronized method, proceeds analogous to that of an object lock for a synchronized instance method. A thread acquires the class lock before it can proceed with the execution of any static synchronized method in the class, blocking other threads wishing to execute any such methods in the same class. This, of course, does not apply to static, non-synchronized methods, which can be invoked at any time. A thread acquiring the lock of a class to execute a static synchronized method, has no bearing on any thread acquiring the lock on any object of the class to execute a synchronized instance method. In other words, synchronization of static methods in a class is independent from the synchronization of instance methods on objects of the class.

A subclass decides whether the new definition of an inherited synchronized method will remain synchronized in the subclass.

Synchronized Blocks

Whereas execution of synchronized methods of an object is synchronized on the lock of the object, the synchronized block allows execution of arbitrary code to be synchronized on the lock of an arbitrary object. The general form of the synchronized statement is as follows:

```
synchronized (<object reference expression>) { <code block> }
```

The <object reference expression> must evaluate to a non-null reference value, otherwise, a NullPointerException is thrown. The code block is usually related to the object on which the synchronization is being done. This is the case with synchronized

methods, where the execution of the method is synchronized on the lock of the current object:

```
public Object pop() {
   synchronized (this) {          // Synchronized block on current object         // ...
```

```
    }
}
```

Once a thread has entered the code block after acquiring the lock on the specified object, no other thread will be able to execute the code block, or any other code requiring the same object lock, until the lock is relinquished. This happens when the execution of the code block completes normally or an uncaught exception is thrown. In contrast to synchronized methods, this mechanism allows fine-grained synchronization of code on arbitrary objects.

Object specification in the synchronized statement is mandatory. A class can choose to synchronize the execution of a part of a method, by using the this reference and putting the relevant part of the method in the synchronized block. The braces of the block cannot be left out, even if the code block has just one statement.

```
class SmartClient {
    BankAccount account;

    // ...

    public void updateTransaction() {

        synchronized (account) {       // (1) synchronized block

 account.update();          // (2)

        }

    }

}
```

In the previous example, the code at (2) in the synchronized block at (1) is synchronized on the BankAccount object. If several threads were to concurrently execute the method updateTransaction() on an object of SmartClient, the statement at (2) would be executed by one thread at a time, only after synchronizing on the BankAccount object associated with this particular instance of SmartClient.

Inner classes can access data in their enclosing context. An inner object might need to synchronize on its associated outer object, in order to ensure integrity of data in the latter. This is illustrated in the following code where the synchronized block at (5) uses the special form of the this reference to synchronize on the outer object associated with an object of the inner class. This setup ensures that a thread executing the method setPi() in an inner object can only access the private double field myPi at (2) in the synchronized block at (5), by first acquiring the lock on the associated outer object. If another thread has the lock of the associated outer object, the thread in the inner object has to wait for the lock to be relinquished before it can proceed with the execution of the synchronized block at (5). However, synchronizing on an inner object and on its associated outer object are independent of each other, unless enforced explicitly, as in the following code:

```
  class Outer {                    // (1) Top-level Class
private double myPi;             // (2)

  protected class Inner {          // (3) Non-static member Class
  public void setPi() {          // (4)
       synchronized(Outer.this) { // (5) Synchronized block on outer object
    myPi = Math.PI;        // (6)
       }
    }

  }

}
```

Synchronized blocks can also be specified on a class lock:

synchronized (<class name>.class) { <code block> }

The block synchronizes on the lock of the object denoted by the reference <class name>.class. A static synchronized method classAction() in class A is equivalent to the following declaration:

```
static void classAction() {
  synchronized (A.class) {     // Synchronized block on class A       // ...
  }
}
```

In summary, a thread can hold a lock on an object

· by executing a synchronized instance method of the object

· by executing the body of a synchronized block that synchronizes on the object

· by executing a synchronized static method of a class

8.5 Thread Transitions

Thread States

Understanding the life cycle of a thread is valuable when programming with threads. Threads can exist in different states. Just because a thread's start() method has been called, it does not mean that the thread has access to the CPU and can start executing straight away. Several factors determine how it will proceed.

Figure 8.3 shows the states and the transitions in the life cycle of a thread.

· Ready-to-run state

A thread starts life in the Ready-to-run state.

· Running state

If a thread is in the Running state, it means that the thread is currently executing.

• Dead state

Once in this state, the thread cannot ever run again.

· Non-runnable states

A running thread can transit to one of the non-runnable states, depending on the circumstances. A thread remains in a non-runnable state until a special transition occurs. A thread does not go directly to the Running state from a non-runnable state, but transits first to the Ready-to-run state. The non-runnable states can be characterized as follows:

o Sleeping: The thread sleeps for a specified amount of time.

o Blocked for I/O: The thread waits for a blocking operation to complete Blocked for join completion: The thread awaits completion of another thread .

o Waiting for notification: The thread awaits notification from another thread

o Blocked for lock acquisition: The thread waits to acquire the lock of an object.

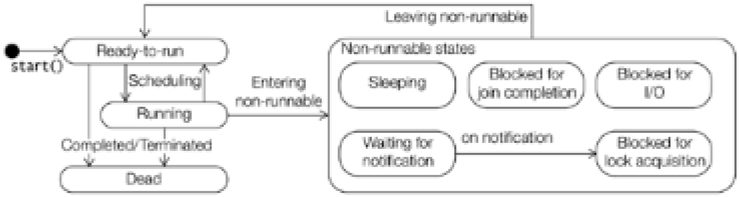


Figure 8.3. Thread States

Various methods from the Thread class are presented next. Examples of their usage are presented in subsequent sections.

final boolean isAlive()

This method can be used to find out if a thread is alive or dead. A thread is alive if it has been started but not yet terminated, that is, it is not in the Dead state.

final int getPriority()
final void setPriority(int newPriority)

The first method returns the priority of the current thread. The second method changes its priority. The priority set will be the minimum of the two values: the specified newPriority and the maximum priority permitted for this thread.

static void yield()

This method causes the current thread to temporarily pause its execution and, thereby, allow other threads to execute.

static void sleep (long millisec) throws InterruptedException

The current thread sleeps for the specified time before it takes its turn at running again.

final void join() throws InterruptedException
final void join(long millisec) throws InterruptedException

A call to any of these two methods invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified time, respectively.

void interrupt()

The method interrupts the thread on which it is invoked. In the Waiting-for-notification, Sleeping, or Blocked-for-join-completion states, the thread will receive an InterruptedException.

Thread Priorities

Threads are assigned priorities that the thread scheduler can use to determine how the threads will be scheduled. The thread scheduler can use thread priorities to determine which thread gets to run. The thread scheduler favors giving CPU time to the thread with the highest priority in the Ready-to-run state. This is not necessarily the thread that has been the longest time in the Ready-to-run state. Heavy reliance on thread priorities for the behavior of a program can make the program unportable across platforms, as thread scheduling is host platform–dependent.

Priorities are integer values from 1 (lowest priority given by the constant Thread. MIN_PRIORITY) to 10 (highest priority given by the constant Thread.MAX_PRIORITY).

The default priority is 5 (Thread.NORM_PRIORITY). A thread inherits the priority of its parent thread. Priority of a thread can be set using the setPriority() method and read using the getPriority() method, both of which are defined in the Thread class. The following code sets the priority of the thread myThread to the minimum of two values: maximum priority and current priority incremented to the next level:

```
myThread.setPriority(Math.min(Thread.MAX_PRIORITY,
myThread.getPriority()+1));
```

Thread Scheduler

Schedulers in JVM implementations usually employ one of the two following strategies:

- Preemptive scheduling.

If a thread with a higher priority than the current running thread moves to the Ready-to-run state, then the current running thread can be preempted (moved to the Ready-to-run state) to let the higher priority thread execute.

· Time-Sliced or Round-Robin scheduling.

A running thread is allowed to execute for a fixed length of time, after which it moves to the Ready-to-run state to await its turn to run again.

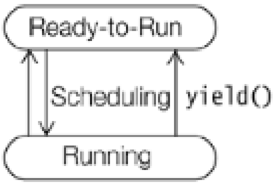
It should be pointed out that thread schedulers are implementation- and platform-dependent; therefore, how threads will be scheduled is unpredictable, at least from platform to platform.

Running and Yielding

After its start() method has been called, the thread starts life in the Ready-to-run state. Once in the Ready-to-run state, the thread is eligible for running, that is, it waits for its turn to get CPU time. The thread scheduler decides which thread gets to run and for how long.

Figure 8.4 illustrates the transitions between the Ready-to-Run and Running states. A call to the static method yield(), defined in the Thread class, will cause the current thread in the Running state to transit to the Ready-to-run state, thus relinquishing the CPU. The thread is then at the mercy of the thread scheduler as to when it will run again. If there are no threads waiting in the Ready-to-run state, this thread continues execution. If there are other threads in the Ready-to-run state, their priorities determine which thread gets to execute.

Figure 8.4. Running and Yielding

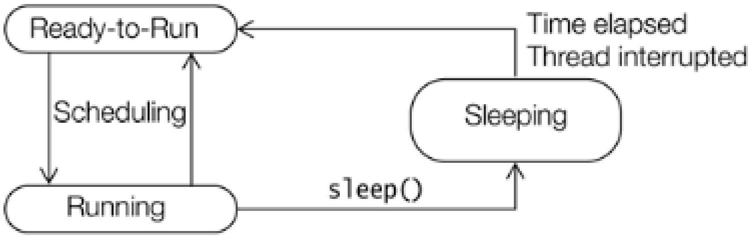


By calling the static method yield(), the running thread gives other threads in the Ready-to-run state a chance to run. A typical example where this can be useful is when a user has given some command to start a CPU-intensive computation, and has the option of canceling it by clicking on a Cancel button. If the computation thread hogs the CPU and the user clicks the Cancel button, chances are that it might take a while before the thread monitoring the user input gets a chance to run and take appropriate action to stop the computation. A thread running such a computation should do the computation in increments, yielding between increments to allow other threads to run. This is illustrated by the following run() method:

```
public void run() {
   try {
      while (!done()) {
         doLittleBitMore();
         Thread.yield();        // Current thread yields        }
   } catch (InterruptedException e) {
        doCleaningUp();
    }
}
```

Sleeping and Waking up

Figure 8.5. Sleeping and Waking up



A call to the static method sleep() in the Thread class will cause the currently running thread to pause its execution and transit to the Sleeping state. The method does not relinquish any lock that the thread might have. The thread will sleep for at least the time specified in its argument, before transitioning to the Ready-to-run state where it takes its turn to run again. If a thread is interrupted while sleeping, it will throw an InterruptedException when it awakes and gets to execute. There are serveral overloaded versions of the sleep() method in the Thread class.

Waiting and Notifying

Waiting and notifying provide means of communication between threads that synchronize on the same object. The threads execute wait() and notify() (or notifyAll()) methods on the shared object for this purpose. These final methods are defined in the Object class, and therefore, inherited by all objects.

These methods can only be executed on an object whose lock the thread holds, otherwise, the call will result in an IllegalMonitorStateException.

```
 final void wait(long timeout) throws InterruptedException
 final void wait(long timeout, int nanos) throws InterruptedException
final void wait() throws InterruptedException
```

A thread invokes the wait() method on the object whose lock it holds. The thread is added to the wait set of the object.

```
final void notify()
final void notifyAll()
```

A thread invokes a notification method on the object whose lock it holds to notify thread(s) that are in the wait set of the object.

Communication between threads is facilitated by waiting and notifying, as illustrated by Figures 8.6 and 8.7. A thread usually calls the wait() method on the object whose lock it holds because a condition for its continued execution was not met. The thread leaves the Running state and transits to the Waiting-for-notification state. There it waits for this condition to occur. The thread relinquishes ownership of the object lock.

Figure 8.6. Waiting and Notifying

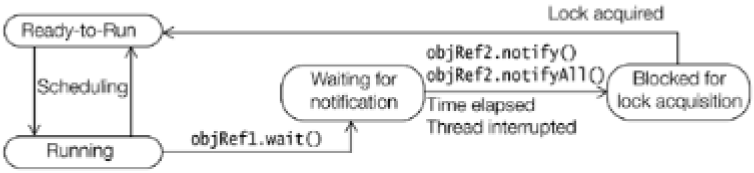
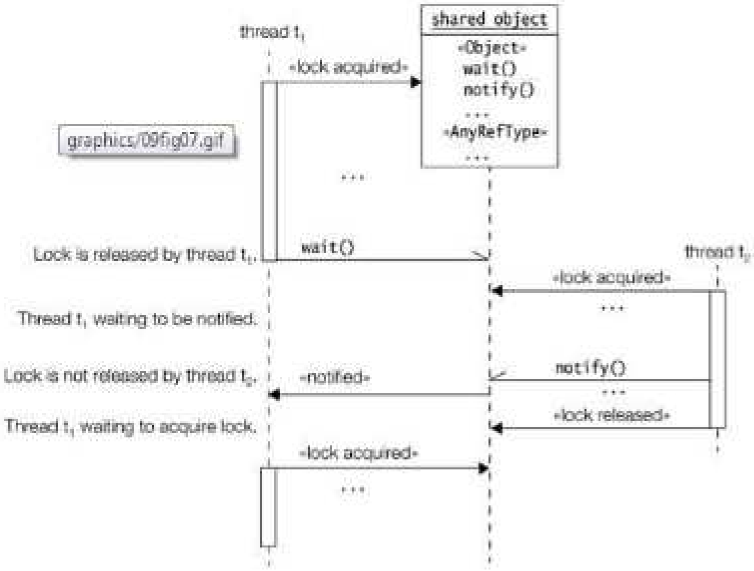


Figure 8.7. Waiting and Notifying



Transition to the Waiting-for-notification state and relinquishing the object lock are completed as one atomic (non-interruptable) operation. The releasing of the lock of the shared object by the thread allows other threads to run and execute synchronized code on the same object after acquiring its lock.

Note that the waiting thread does not relinquish any other object locks that it might hold, only that of the object on which the wait() method was invoked. Objects that have these other locks remain locked while the thread is waiting.

Each object has a wait set containing threads waiting for notification. Threads in the Waiting-for-notification state are grouped according to the object whose wait() method they invoked.

Figure 8.7 shows a thread t1 that first acquires a lock on the shared object, and afterwards invokes the wait() method on the shared object. This relinquishes the object lock and the thread t1 awaits to be notified. While the thread t1 is waiting, another thread t2 can acquire the lock on the shared object for its own purpose.

A thread in the Waiting-for-notification state can be awakened by the occurrence of any one of these three incidents:

1. Another thread invokes the notify() method on the object of the waiting thread, and the waiting thread is selected as the thread to be awakened.
2. The waiting thread times out.
3. Another thread interrupts the waiting thread.

Notify

Invoking the notify() method on an object wakes up a single thread that is waiting on the lock of this object. The selection of a thread to awaken is dependent on the thread policies implemented by the JVM. On being notified, a waiting thread first transits to the Blocked-for-lock-acquisition state to acquire the lock on the object, and not directly to the Ready-to-run state. The thread is also removed from the wait set of the object. Note that the object lock is not relinquished when the notifying thread invokes the notify() method. The notifying thread relinquishes the lock at its own discretion, and the awakened thread will not be able to run until the notifying thread relinquishes the object lock.

When the notified thread obtains the object lock, it is enabled for execution, waiting in the Ready-to-run state for its turn to execute again. Finally, when it does get to execute, the call to the wait() method returns and the thread can continue with its execution.

From Figure 8.7 we see that thread t2 does not relinquish the object lock when it invokes the notify() method. Thread t1 is forced to wait in the Blocked-for-lock-acquisition state. It is shown no privileges and must compete with any other threads waiting for lock acquisition.

A call to the notify() method has no consequences if there are no threads in the wait set of the object.

In contrast to the notify() method, the notifyAll() method wakes up all threads in the wait set of the shared object. They will all transit to the Blocked-for-lock-acquisition state and contend for the object lock as explained earlier.

Time-out

The wait() call specified the time the thread should wait before being timed out, if it was not awakened as explained earlier. The awakened thread competes in the usual manner to execute again. Note that the awakened thread has no way of knowing whether it was timed out or awakened by one of the notification methods.

Interrupt

This means that another thread invoked the interrupt() method on the waiting thread. The awakened thread is enabled as previously explained, but if and when the awakened thread finally gets a chance to run, the return from the wait() call will result in an InterruptedException. This is the reason why the code invoking the wait() method must be prepared to handle this checked exception.

Example 8.4 Waiting and Notifying

```
class StackImpl {
    private Object[] stackArray;
    private volatile int topOfStack;

    StackImpl (int capacity) {
        stackArray = new Object[capacity];
        topOfStack = -1;
    }

    public synchronized Object pop() {
        System.out.println(Thread.currentThread() + ": popping");
   while (isEmpty())
            try {
                System.out.println(Thread.currentThread() + ": waiting to pop");
     wait();                            // (1)
            } catch (InterruptedException e) { }
 Object obj = stackArray[topOfStack];
stackArray[topOfStack--] = null;
        System.out.println(Thread.currentThread() + ": notifying after pop");
    notify();                                  // (2)
        return obj;
    }

    public synchronized void push(Object element) {
System.out.println(Thread.currentThread() + ": pushing");

 while (isFull())

            try {

                System.out.println(Thread.currentThread() + ": waiting to push");
 wait();                          // (3)
        } catch (InterruptedException e) { }
stackArray[++topOfStack] = element;
System.out.println(Thread.currentThread() + ": notifying after push");
  notify();                              // (4)
   }
 public boolean isFull() { return topOfStack >= stackArray.length -1; }
public boolean isEmpty() { return topOfStack < 0; } }


abstract class StackUser extends Thread {
 // (5) Stack user
   protected StackImpl stack;                  // (6)
   StackUser(String threadName, StackImpl stack) {
 super(threadName);
      this.stack = stack;
      System.out.println(this);
      setDaemon(true);                      // (7) Daemon thread
  start();                              // (8) Start this thread.
}
}
class StackPopper extends StackUser {            // (8) Popper
StackPopper(String threadName, StackImpl stack) {
  super(threadName, stack);
   }
   public void run() { while (true) stack.pop(); } }
```

```
class StackPusher extends StackUser {              // (10) Pusher

StackPusher(String threadName, StackImpl stack) {

 super(threadName, stack);
    }
    public void run() { while (true) stack.push(new Integer(1)); } }

public class WaitAndNotifyClient {
    public static void main(String[] args)
           throws InterruptedException {           // (11)
        StackImpl stack = new StackImpl(5);
        new StackPusher("A", stack);
        new StackPusher("B", stack);
        new StackPopper("C", stack);
        System.out.println("Main Thread sleeping.");        Thread.sleep(1000);
        System.out.println("Exit from Main Thread.");     }
}
```

Possible output from the program:

```
Thread[A,5,main]
Thread[B,5,main]
Thread[C,5,main]
Main Thread sleeping.
  ...
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
Thread[B,5,main]: pushing
Thread[B,5,main]: waiting to push
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
Thread[A,5,main]: notifying after push
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
Thread[B,5,main]: waiting to push
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
Thread[A,5,main]: notifying after push
...
Thread[B,5,main]: notifying after push

...

Exit from Main Thread.
```

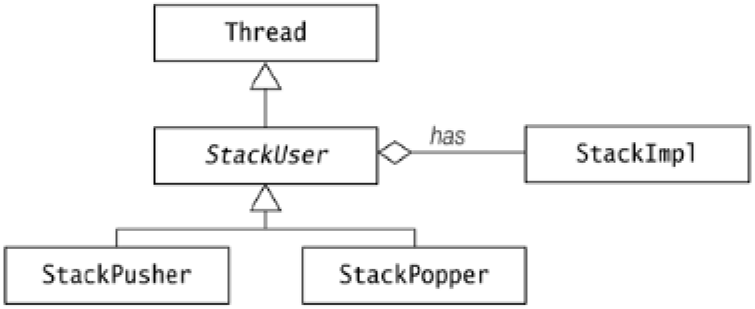
...

In Example 8.4, three threads are manipulating the same stack. Two of them are pushing elements on the stack, while the third one is popping elements off the stack. The class diagram for Example 8.4 is shown in Figure 8.8.

· The subclasses StackPopper at (9) and StackPusher at (10) extend the abstract superclass StackUser at (5).

- Class StackUser, which extends the Thread class, creates and starts each thread.
- Class StackImpl implements the synchronized methods pop() and push().

Figure 8.8: Stack Users



The field topOfStack in class StackImpl is declared volatile, so that read and write operations on this variable will access the master value of this variable, and not any copies, during runtime.

Since the threads manipulate the same stack object and the push() and pop() methods in the class StackImpl are synchronized, it means that the threads synchronize on the same object. In other words, the mutual exclusion of these operations is guaranteed on the same stack object.

Example 8.4 illustrates how a thread waiting as a result of calling the wait() method on an object, is notified by another thread calling the notify() method on the same object, in order for the first thread to start running again.

One usage of the wait() call is shown in Example 8.4 at (1) in the synchronized pop() method. When a thread executing this method on the StackImpl object finds that the stack is empty, it invokes the wait() method in order to wait for some thread to push something on this stack first.

Another use of the wait() call is shown at (3) in the synchronized push() method. When a thread executing this method on the StackImpl object finds that the stack is full, it invokes the wait() method to await some thread removing an element first, in order to make room for a push operation on the stack.

When a thread executing the synchronized method push() on the StackImpl object successfully pushes an element on the stack, it calls the notify() method at (4). The wait set of the StackImpl object contains all waiting threads that had earlier called the wait() method at either (1) or (3) on this StackImpl object. A single thread from the wait set is enabled for running. If this thread was executing a pop operation, it now has a chance of being successful because the stack is not empty at the moment. If this thread was executing a push operation, it can try again to see if there is room on the stack.

When a thread executing the synchronized method pop() on the StackImpl object successfully pops an element off the stack, it calls the notify() method at (2). Again assuming that the wait set of the StackImpl object is not empty, one thread from the set is arbitrarily chosen and enabled. If the notified thread was executing a pop operation, it can proceed to see if the stack still has an element to pop. If the notified thread was executing a push operation, it now has a chance of succeeding because the stack is not full at the moment.

Note that the waiting condition at (1) for the pop operation is executed in a loop. A waiting thread that has been notified is not guaranteed to run straight away. Before it gets to run, another thread may synchronize on the stack and empty it. If the notified thread was waiting to pop the stack, it would now incorrectly pop the stack, because the condition was not tested after notification. The loop ensures that the condition is always tested after notification, sending the thread back to the Waiting-on-notification state if the condition is not met. To avert the analogous danger of pushing on a full stack, the waiting condition at (3) for the push operation is also executed in a loop.

The behavior of each thread can be traced in the output from Example 8.4. Each push-and-pop operation can be traced by a sequence consisting of the name of the operation to be performed, followed by zero or more wait messages, and concluding with a notification after the operation is done. For example, thread A performs two pushes as shown in the output from the program:

```
 Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
...
Thread[A,5,main]: notifying after push
Thread[A,5,main]: pushing
Thread[A,5,main]: waiting to push
...
Thread[A,5,main]: notifying after push
```

Thread B is shown doing one push:

```
Thread[B,5,main]: pushing
Thread[B,5,main]: waiting to push
...
Thread[B,5,main]: notifying after push
```

Whereas thread C pops the stack twice without any waiting:

```
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
...
Thread[C,5,main]: popping
Thread[C,5,main]: notifying after pop
```

When the operations are interweaved, the output clearly shows that the pushers wait when the stack is full, and only push after the stack is popped. The three threads created are daemon threads. Their status is set at (7). They will be terminated if they have not completed when the main user-thread dies, thereby stopping the execution of the program.

Joining

A thread can invoke the overloaded method join() on another thread in order to wait for the other thread to complete its execution before continuing, that is, the first thread waits for the second thread to join it after completion. A running thread t1 invokes the method join() on a thread t2. The join() call has no effect if thread t2 has already completed. If thread t2 is still alive, then thread t1 transits to the Blocked-for-join-completion state. Thread t1 waits in this state until one of these events occur (see Figure 8.9):

- Thread t2 completes.

In this case thread t1 is enabled and when it gets to run, it will continue normally after the join() method call.

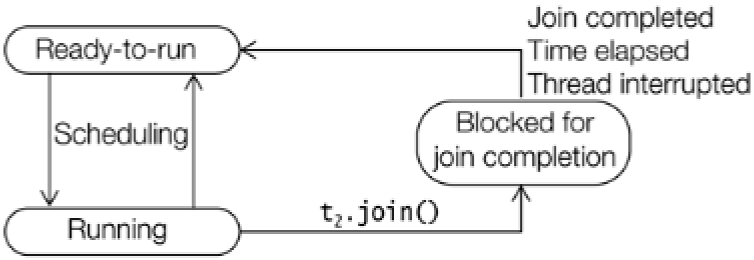
- Thread t1 is timed out.

The time specified in the argument in the join() method call has elapsed, without thread t2 completing. In this case as well, thread t1 is enabled. When it gets to run, it will continue normally after the join() method call.

- Thread t1 is interrupted.

Some thread interrupted thread t1 while thread t1 was waiting for join completion. Thread t1 is enabled, but when it gets to execute, it will now throw an InterruptedException.

Figure 8.9. Joining of Threads



Example 8.5 illustrates joining of threads. The AnotherClient class below uses the Counter class, which extends the Thread class from Example 8.2. It creates two threads that are enabled for execution. The main thread invokes the join() method on the Counter A thread. If the Counter A thread has not already completed, the main thread transits to the Blocked-for-join-completion state. When the Counter A thread completes, the main thread will be enabled for running. Once the main thread is running, it continues with execution after (5). A parent thread can call the isAlive() method to find out whether its child threads are alive, before terminating itself. The call

to the isAlive() method on the Counter A thread at (6) correctly reports that the Counter A thread is not alive. A similar scenario transpires between the main thread and the Counter B thread. The main thread passes through the Blocked-for-join-completion state twice at the most.

Example 8.5 Joining of Threads

```
class Counter extends Thread { /* See Example 8.2. */ }
public class AnotherClient {
   public static void main(String[] args) {

      Counter counterA = new Counter("Counter A");
Counter counterB = new Counter("Counter B");
 try {
         System.out.println("Wait for the child threads to finish.");
   counterA.join();                              // (5)
 if (!counterA.isAlive())                        // (6)
     System.out.println("Counter A not alive.");
  counterB.join();                               // (7)
  if (!counterB.isAlive())                       // (8)
  System.out.println("Counter B not alive.");        }
 catch (InterruptedException e) {
         System.out.println("Main Thread interrupted.");        }

                              System.out.println("Exit from Main Thread.");        }

}
```

Possible output from the program:

```
Thread[Counter A,5,main]
Thread[Counter B,5,main]
Wait for the child threads to finish.
Counter A: 0
Counter B: 0
Counter A: 1
Counter B: 1
Counter A: 2
Counter B: 2
Counter A: 3
Counter B: 3
Counter A: 4
Counter B: 4
Exit from Counter A.
Counter A not alive.
```

```
Exit from Counter B.
Counter B not alive.
Exit from Main Thread.
```

Blocking for I/O

A running thread, on executing a blocking operation requiring a resource (like a call to an I/O method), will transit to the Blocked-for-I/O state. The blocking operation must complete before the thread can proceed to the Ready-to-run state. An example is a thread reading from the standard input terminal, which blocks until input is provided:

```
int input = System.in.read();
```

Thread Termination

A thread can transit to the Dead state from the Running or the Ready-to-run states. The thread dies when it completes its run() method, either by returning normally or by throwing an exception. Once in this state, the thread cannot be resurrected. There is no way the thread can be enabled for running again, not even by calling the start() method once more on the thread object.

Example 8.6 illustrates a typical scenario where a thread can be controlled by one or more threads. Work is performed by a loop body, which the thread executes continually. It should be possible for other threads to start and stop the worker thread. This functionality is implemented by the class Worker at (1), which has a private field theThread declared at (2) to keep track of the Thread object executing its run() method.

The kickStart() method at (3) in class Worker creates and starts a thread if one is not already running. It is not enough to just call the start() method on a thread that has terminated. A new Thread object must be created first. The terminate() method at (4) sets the field theThread to null. Note that this does not affect any Thread object that might have been denoted, by the reference theThread. The runtime system maintains any such Thread object; therefore, changing one of its references does not affect the object.

The run() method at (5) has a loop whose execution is controlled by a special condition. The condition tests to see whether the Thread object denoted by the reference theThread and the Thread object executing now, are one and the same. This is bound to be the case if the reference theThread has the same reference value it was assigned when the thread was created and started in the kickStart() method. The condition will then be true, and the body of the loop will execute. However, if the value in the reference theThread has changed, the condition will be false. In that case, the loop will not execute, the run() method will complete and the thread will terminate.

A client can control the thread implemented by the class Worker, using the kickStart() and the terminate() methods. The client is able to terminate the running thread at the start of the next iteration of the loop body, simply by changing the theThread reference to null.

In Example 8.6, a Worker object is first created at (8) and a thread started on this Worker object at (9). The main thread invokes the yield() method at (10) to temporarily stop its execution, and give the thread of the Worker object a chance to run. The main thread, when it is executing again, terminates the thread of the Worker object at (11), as explained earlier. This simple scenario can be generalized where several threads, sharing a single Worker object, could be starting and stopping the thread of the Worker object.

Example 8.6 Thread Termination

```
class Worker implements Runnable {                          // (1)
 private Thread theThread;                          // (2)
   public void kickStart() {                        // (3)
  if (theThread == null) {
         theThread = new Thread(this);
         theThread.start();
      }
   }

   public void terminate() {                        // (4)
  theThread = null;
   }

   public void run() {                            // (5)
 while (theThread == Thread.currentThread()) {          // (6)
System.out.println("Going around in loops.");         }
   }
}

  public class Controller {
   public static void main(String[] args) {                // (7)
Worker worker = new Worker();                        // (8)
worker.kickStart();                           // (9)
 Thread.yield();                             // (10)
 worker.terminate();                           // (11)     }
}
Possible output from the program:
Going around in loops.
Going around in loops.
Going around in loops.
```

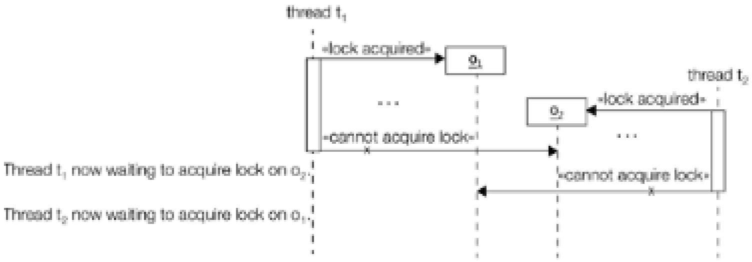
Going around in loops.
Going around in loops.

Deadlocks

A deadlock is a situation where a thread is waiting for an object lock that another thread holds, and this second thread is waiting for an object lock that the first thread holds. Since each thread is waiting for the other thread to relinquish a lock, they both remain waiting forever in the Blocked-for-lock-acquisition state. The threads are said to be deadlocked.

A deadlock is depicted in Figure 8.10. Thread t1 has a lock on object o1, but cannot acquire the lock on object o2. Thread t2 has a lock on object o2, but cannot acquire the lock on object o1. They can only proceed if one of them relinquishes a lock the other one wants, which is never going to happen.

Figure 8.10: DeadLock



The situation in Figure 8.10 is implemented in Example 8.7. Thread t1 at (3) tries to synchronize at (4) and (5), first on string o1 at (1) then on string o2 at (2), respectively. The thread t2 at (6) does the opposite. It synchronizes at (7) and (8), first on string o2 then on string o1, respectively. A deadlock can occur as explained previously.

However, the potential of deadlock in the situation in Example 8.7 is easy to fix. If the two threads acquire the locks on the objects in the same order, then mutual lock dependency is avoided and a deadlock can never occur. This means having the same locking order at (4) and (5) as at (7) and (8). In general, the cause of a deadlock is not always easy to discover, let alone easy to fix.

Example 8.7 Deadlock

```
public class DeadLockDanger {

   String o1 = "Lock " ;                       // (1)
String o2 = "Step ";                           // (2)
   Thread t1 = (new Thread("Printer1") {        // (3)
  public void run() {
          while(true) {
             synchronized(o1) {               // (4)
  synchronized(o2) {           // (5)
System.out.println(o1 + o2);                    }
                }
            }
        }
    });


    Thread t2 = (new Thread("Printer2") {        // (6)
  public void run() {
            while(true) {
                synchronized(o2) {               // (7)
synchronized(o1) {           // (8)
  System.out.println(o2 + o1);                    }
                }
            }
        }
    });
    public static void main(String[] args) {
        DeadLockDanger dld = new DeadLockDanger();         dld.t1.start();
        dld.t2.start();
    }
}
```

Possible output from the program:

```
...
Step Lock
Step Lock
Lock Step
Lock Step
Lock Step
...
```

09

Stream-based IO

9.1 Overview

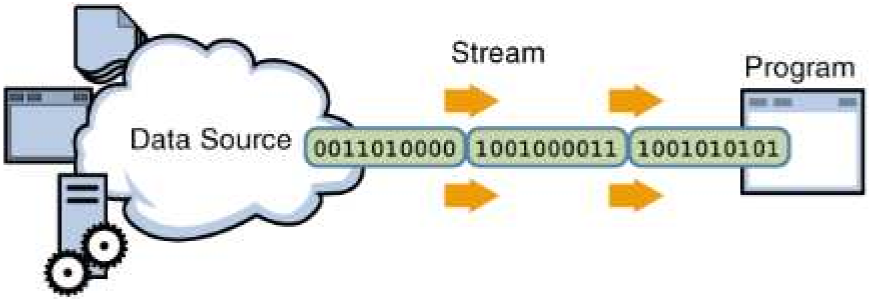
This chapter covers the Java platform classes used for basic I/O. It focuses primarily on I/O Streams, a powerful concept that greatly simplifies I/O operations. The chapter also looks at serialization, which lets a program write whole objects out to streams and read them back again. Then the chapter looks at some file system operations, including random access files. Finally, it touches briefly on the advanced features of the New I/O API. Most of the classes covered are in the java.io package.

9.2 I/O Streams

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

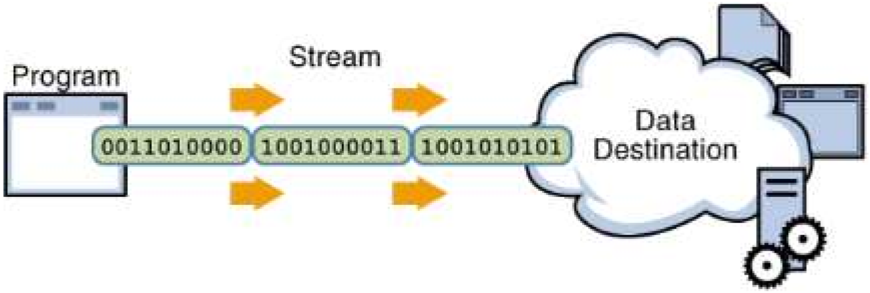
Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:



Reading information into a program.

A program uses an output stream to write data to a destination, one item at time:



Writing information from a program.

In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file xanadu.txt, which contains the following verse:

In Xanadu did Kubla Khan
A stately pleasure-dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.

9.2.1 Byte Streams

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from InputStream and OutputStream.

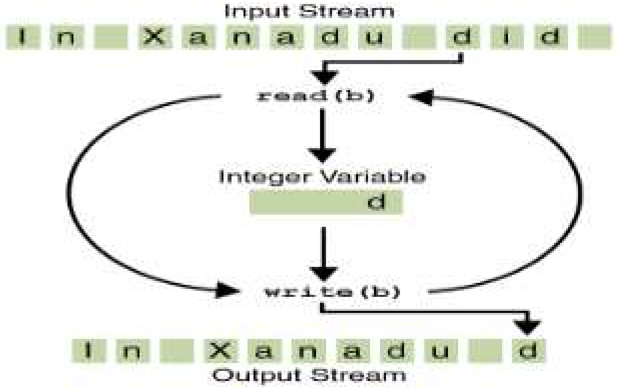
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

Using Byte Streams

We'll explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes {
    public static void main(String[] args) throws IOException {
  FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("xanadu.txt");
out = new FileOutputStream("outagain.txt");
  int c;

            while ((c = in.read()) != -1) {
                out.write(c);
            }

        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



Simple byte stream input and output.

Notice that read() returns an int value. If the input is a stream of bytes, why doesn't read() return a byte value? Using a int as a return type allows read() to use -1 to indicate that it has reached the end of the stream.

Always Close Streams

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks. One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

When Not to Use Byte Streams

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

9.2.2 Character Streams

The Java platform stores character values using Unicode conventions. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII. For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer. If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding. See the Internationalization trail for more information.

Using Character Streams

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
```

```
public class CopyCharacters {
    public static void main(String[] args) throws IOException {
FileReader inputStream = null;
        FileWriter outputStream = null;
    try {
            inputStream = new FileReader("xanadu.txt");
outputStream = new FileWriter("characteroutput.txt");
            int c;
            while ((c = inputStream.read()) != -1) {
 outputStream.write(c);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses FileReader and FileWriter for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

Character Streams that Use Byte Streams

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: InputStreamReader and OutputStreamWriter. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

Line-Oriented I/O

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, BufferedReader and PrintWriter. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The CopyLines example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

```
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.io.IOException;

public class CopyLines {
    public static void main(String[] args) throws IOException {
BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream =   new BufferedReader(new FileReader("xanadu.txt"));
    outputStream =
                new PrintWriter(new FileWriter("characteroutput.txt"));
            String l;
            while ((l = inputStream.readLine()) != null) {
 outputStream.println(l);
            }

        } finally {

            if (inputStream != null) {

                inputStream.close();

            }
            if (outputStream != null) {
                outputStream.close();
            }
        }
    }
}
```

Invoking readLine returns a line of text with the line CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

9.2.3 Buffered Streams

Most of the examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full. A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

```
            inputStream =
            new BufferedReader(new FileReader("xanadu.txt"));
        outputStream =
            new BufferedWriter(new FileWriter("characteroutput.txt"));
```

There are four buffered stream classes used to wrap unbuffered streams: BufferedInputStream and BufferedOutputStream create buffered byte streams, while BufferedReader and BufferedWriter create buffered character streams.

Flushing Buffered Streams

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer. Some buffered output classes support autoflush, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format.

To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

9.2.4 Scanning and Formatting

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with bits of data. The formatting API assembles data into nicely formatted, human-readable form.

Scanning

Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

Breaking Input into Tokens

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators. For the full list, refer to the documentation for Character.isWhitespace.) To see how scanning works, let's look at ScanXan, a program that reads the individual words in xanadu.txt and prints them out, one per line.

```
 import java.io.*;
import java.util.Scanner;
 public class ScanXan {
   public static void main(String[] args) throws IOException {
 Scanner s = null;
        try {
            s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));
         while (s.hasNext()) {
                System.out.println(s.next());
            }
        } finally {
            if (s != null) {
                s.close();
            }
        }
    }
}
```

Notice that ScanXan invokes Scanner's close method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of ScanXan looks like this:

```
In
Xanadu
did
Kubla
Khan
A
stately
pleasure-dome
...
```

To use a different token separator, invoke useDelimiter(), specifying a regular expression. For example, suppose you wanted the token separator to be a comma, optionally followed by white space. You would invoke, s.useDelimiter(",\\s*");

Translating Individual Tokens

The ScanXan example treats all input tokens as simple String values. Scanner also supports tokens for all of the Java language's primitive types (except for char), as well as BigInteger and BigDecimal. Also, numeric values can use thousands separators. Thus, in a US locale, Scanner correctly reads the string "32,767" as representing an integer value.

We have to mention the locale, because thousands separators and decimal symbols are locale specific. So, the following example would not work correctly in all locales if we didn't specify that the scanner should use the US locale. That's not something you usually have to worry about, because your input data usually comes from sources that use the same locale as you do. But this example is part of the Java Tutorial and gets distributed all over the world.

The ScanSum example reads a list of double values and adds them up. Here's the source:

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.Scanner;
import java.util.Locale;
```

```
   public class ScanSum {
  public static void main(String[] args) throws IOException {
Scanner s = null;
    double sum = 0;
    try {
      s = new Scanner(
          new BufferedReader(new FileReader("usnumbers.txt")));
s.useLocale(Locale.US);

while (s.hasNext()) {
        if (s.hasNextDouble()) {
            sum += s.nextDouble();
          } else {
            s.next();
          }
      }
    } finally {
      s.close();
    }

    System.out.println(sum);

  }

}
```

And here's the sample input file, usnumbers.txt 8.5

32,767

3.14159

1,000,000.1

The output string is "1032778.74159". The period will be a different character in some locales, because System.out is a PrintStream object, and that class doesn't provide a way to override the default locale. We could override the locale for the whole program — or we could just use formatting,

Formatting

Stream objects that implement formatting are instances of either PrintWriter, a character stream class, and PrintStream, a byte stream class.

Note: The only PrintStream objects you are likely to need are System.out and System.err. (See I/O from the Command Line for more on these objects.) When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream.

Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output. In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

- print and println format individual values in a standard way.
- format formats almost any number of values based on a format string, with many options for precise formatting.

The print and println Methods

Invoking print or println outputs a single value after converting the value using the appropriate toString method. We can see this in the Root example:

```
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);
         System.out.print("The square root of ");         System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");     }
}
```

Here is the output of Root:

```
 The square root of 2 is 1.4142135623730951.
 The square root of 5 is 2.23606797749979.
```

The i and r variables are formatted twice: the first time using code in an overload of print, the second time by conversion code automatically generated by the Java compiler, which also utilizes toString. You can format any value this way, but you don't have much control over the results.

The format Method

The format method formats multiple arguments based on a format string. The format string consists of static text embedded with format specifiers; except for the format specifiers, the format string is output unchanged.

```
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);     }
}
```

Here is the output:

```
The square root of 2 is 1.414214.
```

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:

- d formats an integer value as a decimal value.
- f formats a floating point value as a decimal value.
- n outputs a platform-specific line terminator.

Here are some other conversions:

- x formats an integer as a hexadecimal value.
- s formats any value as a string.
- tB formats an integer as a locale-specific month name.

There are many other conversions.

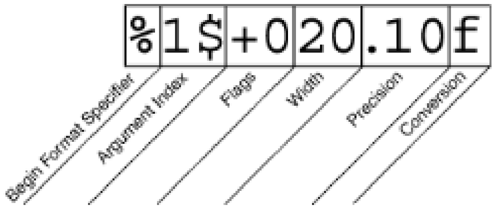
Note: Except for %% and %n, all format specifiers must match an argument. If they don't, an exception is thrown.
In the Java programming language, the \n escape always generates the linefeed character (\u000A). Don't use \n unless you specifically want a linefeed character. To get the correct line separator for the local platform, use %n.
In addition to the conversion, a format specifier can contain several additional elements that further customize the formatted output. Here's an example, Format, that uses every possible kind of element.

```
public class Format {
    public static void main(String[] args) {
        System.out.format("%f, %1$+020.10f %n", Math.PI);     }
}
```

Here's the output:

```
3.141593, +00000003.1415926536
```

The additional elements are all optional. The following figure shows how the longer specifier breaks down into elements.



Elements of a Format Specifier.

The elements must appear in the order shown. Working from the right, the optional elements are:

- **Precision.** For floating point values, this is the mathematical precision of the formatted value. For s and other general conversions, this is the maximum width of the formatted value; the value is right-truncated if necessary.

- **Width**. The minimum width of the formatted value; the value is padded if necessary. By default the value is left-padded with blanks.

- **Flags** specify additional formatting options. In the Format example, the + flag specifies that the number should always be formatted with a sign, and the 0 flag specifies that 0 is the padding character. Other flags include - (pad on the right) and , (format number with locale-specific thousands separators). Note that some flags cannot be used with certain other flags or with certain conversions.

- **The Argument** Index allows you to explicitly match a designated argument. You can also specify < to match the same argument as the previous specifier. Thus the example could have said:

```
System.out.format("%f, %<+020.10f %n", Math.PI);
```

9.2.5 I/O from the Command Line

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

Standard Streams

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. For more information, refer to the documentation for your command line interpreter.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as PrintStream objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader. InputStreamReader cin = new InputStreamReader(System.in);

The Console

A more advanced alternative to the Standard Streams is the Console. This is a single, predefined object of type Console that has most of the features provided by the Standard Streams, and others besides. The Console is particularly useful for secure password entry. The Console object also provides input and output streams that are true character streams, through its reader and writer methods.

Before a program can use the Console, it must attempt to retrieve the Console object by invoking System.console(). If the Console object is available, this method returns it. If System.console returns NULL, then Console operations are not permitted, either because the OS doesn't support them or because the program was launched in a noninteractive environment.

The Console object supports secure password entry through its readPassword method. This method helps secure password entry in two ways. First, it suppresses echoing, so the password is not visible on the user's screen. Second, readPassword returns a character array, not a String, so the password can be overwritten, removing it from memory as soon as it is no longer needed.

The Password example is a prototype program for changing a user's password. It demonstrates several Console methods.

```
 import java.io.Console;
import java.util.Arrays;
import java.io.IOException;
 public class Password {
    public static void main (String args[]) throws IOException {
        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }

        String login = c.readLine("Enter your login: ");
char [] oldPassword = c.readPassword("Enter your old password: ");
        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 =
                    c.readPassword("Enter your new password: ");
   char [] newPassword2 =
                    c.readPassword("Enter new password again: ");
 noMatch = ! Arrays.equals(newPassword1, newPassword2);
 if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");                 }
 else {
                                                            change(login, newPassword1);


                    c.format("Password for %s changed.%n", login);

  }

                Arrays.fill(newPassword1, ' ');

                Arrays.fill(newPassword2, ' ');

            } while (noMatch);

        }



        Arrays.fill(oldPassword, ' ');
    }

    //Dummy verify method.
    static boolean verify(String login, char[] password) {
 return true;
    }

    //Dummy change method.
    static void change(String login, char[] password) {} }
```

Password follows these steps:

1. Attempt to retrieve the Console object. If the object is not available, abort.
2. Invoke Console.readLine to prompt for and read the user's login name.
3. Invoke Console.readPassword to prompt for and read the user's existing password.
4. Invoke verify to confirm that the user is authorized to change the password. (In this example, verify is a dummy method that always returns true.)
5. Repeat the following steps until the user enters the same password twice:
    a. Invoke Console.readPassword twice to prompt for and read a new password.
    b. If the user entered the same password both times, invoke change to change it.

    (Again, change is a dummy method.)

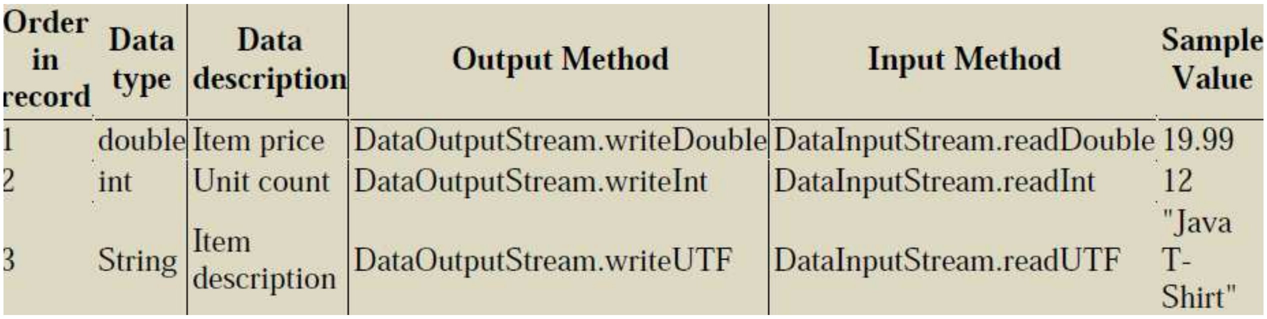
    c. Overwrite both passwords with blanks.

6. Overwrite the old password with blanks.

9.2.6 Data Streams

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the DataInput interface or the DataOutput interface. This section focuses on the most widely-used implementations of these interfaces, DataInputStream and DataOutputStream.

The DataStreams example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:



Order in record	Data type	Data description	Output Method	Input Method	Sample Value
1	double	Item price	DataOutputStream.writeDouble	DataInputStream.readDouble	19.99
2	int	Unit count	DataOutputStream.writeInt	DataInputStream.readInt	12
3	String	Item description	DataOutputStream.writeUTF	DataInputStream.readUTF	"Java T-Shirt"

Let's examine crucial code in DataStreams. First, the program defines some constants containing the name of the data file and the data that will be written to it:

```
 static final String dataFile = "invoicedata";



static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = { "Java T-shirt",
        "Java Mug",
        "Duke Juggling Dolls",
        "Java Pin",
        "Java Key Chain" }
;
```

Then DataStreams opens an output stream. Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

```
 out = new DataOutputStream(new
            BufferedOutputStream(new FileOutputStream(dataFile)));
```

DataStreams writes out the records and closes the output stream.

```
for (int i = 0; i < prices.length; i ++) {
    out.writeDouble(prices[i]);
    out.writeInt(units[i]);
    out.writeUTF(descs[i]);
}
```

The writeUTF method writes out String values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now DataStreams reads the data back in again. First it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));
double price;
int unit;
String desc;
double total = 0.0;
```

Now DataStreams can read each record in the stream, reporting on the data it encounters.

```
try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("You ordered %d units of %s at $%.2f%n",
unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
}
```

Notice that DataStreams detects an end-of-file condition by catchin EOFException, instead of testing for an invalid return value. All implementations of DataInput methods use EOFException instead of return values.

Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.

The correct type to use for currency values is java.math.BigDecimal. Unfortunately, BigDecimal is an object type, so it won't work with data streams. However, BigDecimal will work with object streams

9.2.7 Object Streams

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

The object stream classes are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are subinterfaces of DataInput and DataOutput.

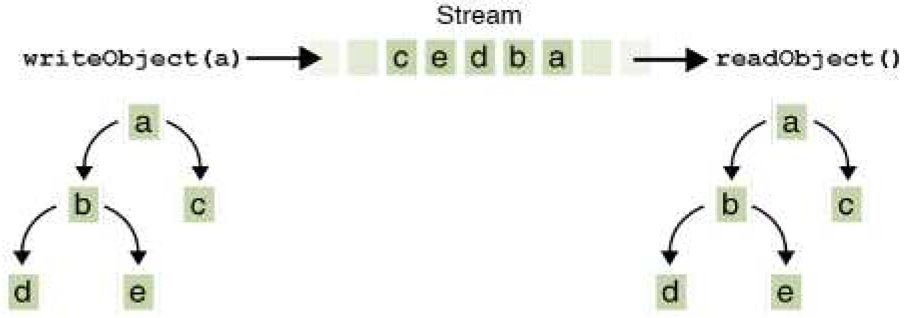
That means that all the primitive data I/O methods covered in DataStreams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values. The ObjectStreams example illustrates this. ObjectStreams creates the same application as DataStreams, with a couple of changes. First, prices are now BigDecimal objects, to better represent fractional values. Second, a Calendar object is written to the data file, indicating an invoice date.

If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a ClassNotFoundException. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding ClassNotFoundException to the main method's throws clause.

Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named a. This object contains references to objects b and c, while b contains references to d and e. Invoking writeobject(a) writes not just a, but all the objects necessary to reconstitute a, so the other four objects in this web are written also. When a is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.



I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

```
 Object ob1 = in.readObject();

Object ob2 = in.readObject();
```

This results in two variables, ob1 and ob2, that are references to a single object. However, if a single object is written to two different streams, it is effectively duplicated — a single program reading both streams back will see two distinct objects.

9.3 File I/O

So far, this lesson has focused on streams, which provide a simple model for reading and writing data. Streams work with a large variety of data sources and destinations, including disk files. However, streams don't support all the operations that are common with disk files. In this part of the lesson, we'll focus on non-stream file I/O. There are two topics:

- **File** is a class that helps you write platform-independent code that examines and manipulates files and directories.
- **Random access files** support nonsequential access to disk file data.

9.3.1 File Objects

The File class makes it easier to write platform-independent code that examines and manipulates files. The name of this class is misleading: File instances represent file names, not files. The file corresponding to the file name might not even exist.

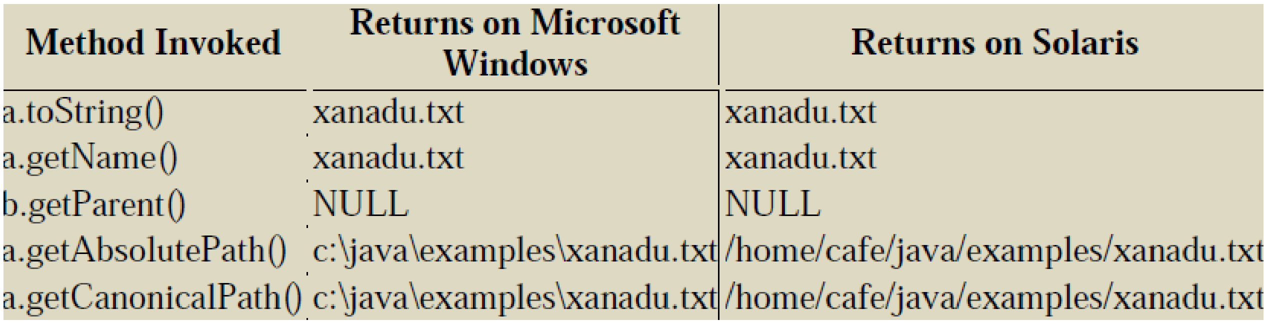
Why create a File object for a file that doesn't exist? A program can use the object to parse a file name. Also, the file can be created by passing the File object to the constructor of some classes, such as FileWriter. If the file does exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, or changing its permissions.

A File Has Many Names

A File object contains the file name string used to construct it. That string never changes throughout the lifetime of the object. A program can use the File object to obtain other versions of the file name, some of which may or may not be the same as the original file name string passed to the constructor. Suppose a program creates a File object with the constructor invocation

File a = new File("xanadu.txt");

The program invokes a number of methods to obtain different versions of the file name. The program is then run both on a Microsoft Windows system (in directory c:\java\examples) and a Solaris system (in directory /home/cafe/java/examples). Here is what the methods would return:

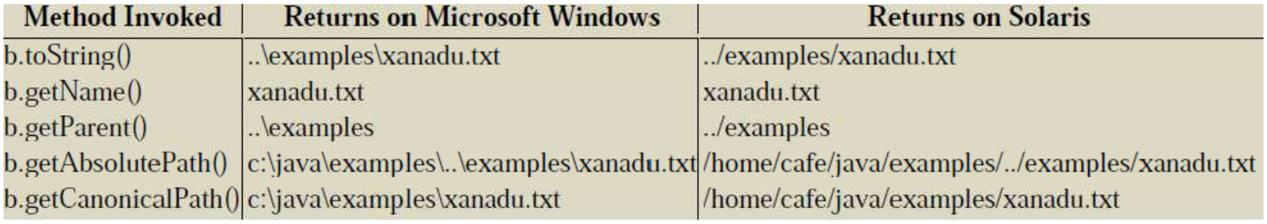


Method Invoked	Returns on Microsoft Windows	Returns on Solaris
a.toString()	xanadu.txt	xanadu.txt
a.getName()	xanadu.txt	xanadu.txt
b.getParent()	NULL	NULL
a.getAbsolutePath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt
a.getCanonicalPath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt

Then the same program constructs a File object from a more complicated file name, using File.separator to specify the file name in a platform-independent way.

File b = new File(".." + File.separator + "examples" + File.separator + "xanadu.txt");

Although b refers to the same file as a, the methods return slightly different values:



Method Invoked	Returns on Microsoft Windows	Returns on Solaris
b.toString()	..\examples\xanadu.txt	../examples/xanadu.txt
b.getName()	xanadu.txt	xanadu.txt
b.getParent()	..\examples	../examples
b.getAbsolutePath()	c:\java\examples\..\examples\xanadu.txt	/home/cafe/java/examples/../examples/xanadu.txt
b.getCanonicalPath()	c:\java\examples\xanadu.txt	/home/cafe/java/examples/xanadu.txt

Running the same program on a Linux system would give results similar to those on the Solaris system.

It's worth mentioning that File.compareTo() would not consider a and b to be the same. Even though they refer to the same file, the names used to construct them are different.

The FileStuff example creates File objects from names passed from the command line and exercises various information methods on them. You'll find it instructive to run FileStuff on a variety of file names. Be sure to include directory names as well as the names of files that don't actually exist. Try passing FileStuff a variety of relative and absolute path names.

Manipulating Files

If a File object names an actual file, a program can use it to perform a number of useful operations on the file. These include passing the object to the constructor for a stream to open the file for reading or writing. The delete method deletes the file immediately, while the deleteOnExit method deletes the file when the virtual machine terminates.

The setLastModified sets the modification date/time for the file. For example, to set the modification time of xanadu.txt to the current time, a program could do new File("xanadu.txt").setLastModified(new Date().getTime());

The renameTo() method renames the file. Note that the file name string behind the File object remains unchanged, so the File object will not refer to the renamed file.

Working with Directories

File has some useful methods for working with directories. The mkdir method creates a directory. The mkdirs method does the same thing, after first creating any parent directories that don't yet exist.

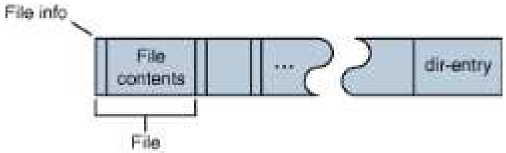
The list and listFiles methods list the contents of a directory. The list method returns an array of String file names, while listFiles returns an array of File objects.

Static Methods

File contains some useful static methods.
The createTempFile method creates a new file with a unique name and returns a File object referring to it
The listRoots returns a list of file system root names. On Microsoft Windows, this will be the root directories of mounted drives, such as a:\ and c:\. On UNIX and Linux systems, this will be the root directory, /.

9.3.2 Random Access Files

Random access files permit nonsequential, or random, access to a file's contents. Consider the archive format known as ZIP. A ZIP archive contains files and is typically compressed to save space. It also contains a directory entry at the end that indicates where the various files contained within the ZIP archive begin, as shown in the following figure.



A ZIP archive.

Suppose that you want to extract a specific file from a ZIP archive. If you use a sequential access stream, you have to:

Open the ZIP archive.

Search through the ZIP archive until you locate the file you want to extract.
Extract the file.

Close the ZIP archive.

Using this procedure, on average, you'd have to read half the ZIP archive before finding the file that you want to extract. You can extract the same file from the ZIP archive more efficiently by using the seek feature of a random access file and following these steps:

Open the ZIP archive.

Seek to the directory entry and locate the entry for the file you want to extract from the ZIP archive.

Seek (backward) within the ZIP archive to the position of the file to extract.

Extract the file.

Close the ZIP archive.

This algorithm is more efficient because you read only the directory entry and the file that you want to extract.

The java.io.RandomAccessFile class implements both the DataInput and DataOutput interfaces and therefore can be used for both reading and writing. RandomAccessFile is similar to FileInputStream and FileOutputStream in that you specify a file on the native file system to open when you create it. When you create a RandomAccessFile, you must indicate whether you will be just reading the file or also writing to it. (You have to be able to read a file in order to write it.) The following code creates a RandomAccessFile to read the file named xanadu.txt:

```
new RandomAccessFile("xanadu.txt", "r");
```

And this one opens the same file for both reading and writing: new RandomAccessFile("xanadu.txt", "rw");

After the file has been opened, you can use the common read or write methods defined in the DataInput and DataOutput interfaces to perform I/O on the file. RandomAccessFile supports the notion of a file pointer. The file pointer indicates the current location in the file. When the file is first created, the file pointer is set to 0, indicating the beginning of the file. Calls to the read and write methods adjust the file pointer by the number of bytes read or written.

In addition to the normal file I/O methods that implicitly move the file pointer when the operation occurs, RandomAccessFile contains three methods for explicitly manipulating the file pointer.

- int skipBytes(int) − Moves the file pointer forward the specified number of bytes
- void seek(long) − Positions the file pointer just before the specified byte
- long getFilePointer() − Returns the current byte location of the file pointer

10

JAVA Generics

1. Introduction

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of generics.

This chapter is aimed at introducing you to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll soon see that there are both similarities and important differences.

Generics allows us to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.

Here is a typical usage of that sort:

```
 List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an Object will be returned by the iterator. To ensure the assignment to a variable of type Integer is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer might be mistaken. What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics.

Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable myIntList. It specifies that this is not just an arbitrary List, but a List of Integer, written List<Integer>.

We say that List is a generic interface that takes a type parameter - in this case, Integer. We also specify a type parameter when creating the list object.

The other thing to pay attention to is that the cast is gone on line 3’. Now, you might think that all we’ve accomplished is to move the clutter around.

Instead of a cast to Integer on line 3, we have Integer as a type parameter on line 1’. However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that myIntList is declared with type List<Integer>, this tells us something about the variable myIntList, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

2 Defining Simple Generics

Here is a small excerpt from the definitions of the interfaces List and Iterator in package

java.util:

```
public interface List<E> {
void add(E x);
Iterator<E> iterator();
}
public interface Iterator<E> {
E next();
boolean hasNext();
}
```

This should all be familiar, except for the stuff in angle brackets. Those are the declarations of the formal type parameters of the interfaces List and Iterator.

Type parameters can be used throughout the generic declaration, pretty much where you would use ordinary types. In the introduction, we saw invocations of the generic type declaration List, such as List<Integer>. In the invocation (usually called a parameterized type), all occurrences of the formal type parameter (E in this case) are replaced by the actual type argument (in this case, Integer).

You might imagine that List<Integer> stands for a version of List where E has been uniformly replaced by Integer:

```
public interface IntegerList {
void add(Integer x)
Iterator<Integer> iterator();

}
```

This intuition can be helpful, but it's also misleading. It is helpful, because the parameterized type List<Integer> does indeed have methods that look just like this expansion. It is misleading, because the declaration of a generic is never actually expanded in this way. There aren't multiple copies of the code: not in source, not in binary, not on disk and not in memory.

If you are a C++ programmer, you'll understand that this is very different than a C++ template. A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration. Type parameters are analogous to the ordinary parameters used in methods or constructors. Much like a method has formal value parameters that describe the kinds of values it operates on, a generic declaration has formal type parameters.

When a method is invoked, actual arguments are substituted for the formal parameters, and the method body is evaluated. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.

A note on naming conventions:

We recommend that you use pithy (single character if possible) yet evocative names for formal type parameters. It's best to avoid lower case characters in those names, making it easy to distinguish formal type parameters from ordinary classes and interfaces. Many container types use E, for element, as in the examples above. We'll see some additional conventions in later examples.

3 Generics and Subtyping

Let's test our understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a List of String a List of Object. Most people's instinct is to answer:

"sure!".

Well, take a look at the next few lines:

```
lo.add(new Object()); // 3
```

```
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

Here we’ve aliased ls and lo. Accessing ls, a list of String, through the alias lo, we can insert arbitrary objects into it. As a result ls does not hold just Strings anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error. In general, if Foo is a subtype (subclass or subinterface) of Bar, and G is some generic type declaration, it is not the case that G<Foo> is a subtype of G<Bar>.

This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions. The problem with that intuition is that it assumes that collections don’t change. Our instinct takes these things to be immutable. For example, if the department of motor vehicles supplies a list of drivers to the census bureau, this seems reasonable.

We think that a List<Driver> is a List<Person>,
assuming that Driver is a subtype of Person.

In fact, what is being passed is a copy of the registry of drivers. Otherwise, the census bureau could add new people who are not drivers into the list, corrupting the DMV’ s records. In order to cope with this sort of situation, it’ s useful to consider more flexible generic types. The rules we’ ve seen so far are quite restrictive.

4 Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection.

Here’s how you might write it in an older version of the language:

```
void printCollection(Collection c) {
Iterator i = c.iterator();
for (k = 0; k < c.size(); k++) {
System.out.println(i.next());
}
}
```

And here is a naive attempt at writing it using generics (and the new for loop syntax):

```
                         void printCollection(Collection<Object> c) {
                                      for (Object e : c) {
System.out.println(e);
}
}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes

Collection<Object>, which, as we’ve just demonstrated, is not a supertype of all kinds of collections!

So what is the supertype of all kinds of collections? It’ s written Collection<?> (pronounced “collection of unknown”) , that is, a collection whose element type matches anything. It’ s called a wildcard type for obvious reasons.

We can write:

```
void printCollection(Collection<?> c) {
for (Object e : c) {
System.out.println(e);
}
}
```

and now, we can call it with any type of collection. Notice that inside printCollection(),

we can still read elements from c and give them type Object. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn’t safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();

c.add(new Object()); // compile time error
```

Since we don’t know what the element type of c stands for, we cannot add objects to it. The add() method takes arguments of type E, the element type of the collection. When the actual type parameter is ?, it stands for some unknown type. Any parameter we pass to add would have to be a subtype of this unknown type. Since we don’t know what type that is, we cannot pass anything in. The sole exception is null, which is a member of every type.

On the other hand, given a List<?>, we can call get() and make use of the result. The result type is an unknown type, but we always know that it is an object. It is therefore safe to assign the result of get() to a variable of type Object or pass it as a parameter where the type Object is expected.

4.1 Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {

public abstract void draw(Canvas c);

}
```

```
public class Circle extends Shape {
private int x, y, radius;
public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
private int x, y, width, height;
public void draw(Canvas c) { ... }
}
```

These classes can be drawn on a canvas:

```
public class Canvas {
public void draw(Shape s) {
s.draw(this);
}
}
```

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in Canvas that draws them all:

```
public void drawAll(List<Shape> shapes) {
      for (Shape s: shapes) {
   s.draw(this);
   }
}
```

Now, the type rules say that drawAll() can only be called on lists of exactly Shape: it cannot, for instance, be called on a List<Circle>. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a List<Circle>. What we really want is for the method to accept a list of any kind of shape:

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

There is a small but very important difference here: we have replaced the type

List<Shape> with List<? **extends** Shape>.

Now drawAll() will accept lists of any subclass of Shape, so we can now call it on a List<Circle> if we want. List<? extends Shape> is an example of a bounded wildcard. The ? stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of Shape1. We say that Shape is the upper bound of the wildcard.

There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into shapes in the body of the method. For instance, this is not allowed:

```
public void addRectangle(List<? extends Shape> shapes) {

shapes.add(0, new Rectangle()); // compile-time error! }
```

You should be able to figure out why the code above is disallowed. The type of the second parameter to shapes.add() is ? extends Shape - an unknown subtype of Shape. Since we don’ t know what type it is, we don’ t know if it is a supertype of Rectangle; it might or might not be such a supertype, so it isn’ t safe to pass a Rectangle there.

Bounded wildcards are just what one needs to handle the example of the DMV passing its data to the census bureau. Our example assumes that the data is represented by mapping from names (represented as strings) to people (represented by reference types such as Person or its subtypes, such as Driver). Map<K,V> is an example of a generic type that takes two type arguments, representing the keys and values of the map.

Again, note the naming convention for formal type parameters - K for keys and V for values.

```
public class Census {
public static void
addRegistry(Map<String, ? extends Person> registry) { ...} }...
Map<String, Driver> allDrivers = ...;
Census.addRegistry(allDrivers);
```

5 Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection.

Here is a first attempt:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
for (Object o : a) {
c.add(o); // compile time error
}}
```

By now, you will have learned to avoid the beginner’s mistake of trying to use Collection<Object> as the type of the collection parameter. You may or may not have recognized that using Collection<?> isn’t going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use generic methods. Just like type declarations, method declarations can be generic - that is, parameterized by one or more type parameters.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
 for (T o : a) {
c.add(o); // correct
}}
```

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

```
 Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferred to be Number
 fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```

Notice that we don’t have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let’s examine a few methods from the Collection libraries.

```
 interface Collection<E> {
public boolean containsAll(Collection<?> c);
public boolean addAll(Collection<? extends E> c); }
```

We could have used generic methods here instead:

```
interface Collection<E> {
public <T> boolean containsAll(Collection<T> c);
public <T extends E> boolean addAll(Collection<T> c); // hey, type variables can have bounds too!
}
```

However, in both containsAll and addAll, the type parameter T is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used. It is possible to use both generic methods and wildcards in tandem. Here is the method Collections.copy():

```
class Collections {

public static <T> void copy(List<T> dest, List<? extends T> src){...} }
```

Note the dependency between the types of the two parameters. Any object copied from the source list, src, must be assignable to the element type T of the destination list, dst. So the element type of src can be any subtype of T - we don't care which.

The signature of copy expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter. We could have written the signature for this method another way, without using wildcards at all:

```
class Collections {

public static <T, S extends T>

void copy(List<T> dest, List<S> src){...}

}
```

This is fine, but while the first type parameter is used both in the type of dst and in the bound of the second type parameter, S, S itself is only used once, in the type of src - nothing else depends on it. This is a sign that we can replace S with a wildcard.

Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible. Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class Shape, and have drawAll() store its incoming argument into the history field.

```
static List<List<? extends Shape>> history =
new ArrayList<List<? extends Shape>>();

 public void drawAll(List<? extends Shape> shapes) {

history.addLast(shapes);

for (Shape s: shapes) {

s.draw(this);

}}
```

Finally, again let’s take note of the naming convention used for the type parameters. We use T for type, whenever there isn’t anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor T in the alphabet, such as S. If a generic method appears inside a generic class, it’s a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

11

Collection Framework

This chapter describes the Java Collections Framework. Here you will learn what collections are and how they can make your job easier and programs better.You'll learn about the core elements—interfaces, implementations, and algorithms—that comprise the Java Collections Framework.

Introduction to Collections

A collection—sometimes called a container—is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

If you've used the Java programming language—or just about any other programming language—you're already familiar with collections. Collection implementations in earlier (pre-1.2) versions of the Java platform included Vector,1 Hashtable,2 and array.3 However, those earlier versions did not contain a collections framework.

What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

Interfaces These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.

Implementations These are the concrete implementations of the collection interfaces.

In essence, they are reusable data structures.

Algorithms These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Benefits of the Java Collections Framework

Reduces Programming Effort

By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level “plumbing” required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.

Increases Program Speed and Quality

This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you’re freed from the drudgery of writing your own data structures, you’ll have more time to devote to improving programs’ quality and performance.

Allows Interoperability among Unrelated APIs

The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.

Reduces Effort to Learn and to Use New APIs

Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.

Reduces Effort to Design New APIs

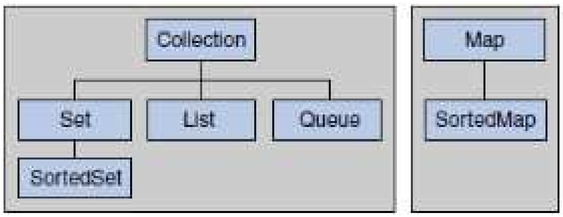
This is the flip side of the previous advantage. Designers and implementers don’t have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

Fosters Software Reuse

New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Interfaces

The core collection interfaces encapsulate different types of collections, which are shown in the following figure. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the figure below, the core collection interfaces form a hierarchy.



The core collection interfaces.

A Set is a special kind of Collection; a SortedSet is a special kind of Set, and so forth. Note also that the hierarchy consists of two distinct trees—a Map is not a true Collection.

Note that all the core collection interfaces are generic. For example, this is the declaration of the Collection interface:

public interface Collection<E>...

The <E> syntax tells you that the interface is generic. When you declare a Collection instance, you can and should specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

When you understand how to use these interfaces, you will know most of what there is to know about the Java Collections Framework. This section discusses general guidelines for effective use of the interfaces, including when to use which interface. You'll also learn programming idioms for each interface to help you get the most out of it.

To keep the number of core collection interfaces manageable, the Java platform doesn't provide separate interfaces for each variant of each collection type. (Such variants might include immutable, fixed-size, and append-only.) Instead, the modification operations in each interface are designated optional—a given implementation may elect not to support all operations. If an unsupported operation is invoked, a collection throws an UnsupportedOperationException.4 Implementations are responsible for documenting which of the optional operations they support. All of the Java platform's general-purpose implementations support all of the optional operations.

The following list describes the core collection interfaces:

Collection The root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn’t provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

Set A collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student’s schedule, or the processes running on a machine.

List An ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you’ve used Vector, you’re familiar with the general flavour of List.

Queue A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements’ natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

Map An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you’ve used Hashtable, you’re already familiar with the basics of Map.

The last two core collection interfaces are merely sorted versions of Set and Map:

SortedSet A Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

SortedMap A Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

The Collection Interface

A Collection represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a conversion constructor, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection’s subinterface or implementation type. In other words, it allows you to convert the collection’s type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c:

```
List<String> list = new ArrayList<String>(c);
```

The following shows the Collection interface:

```
public interface Collection<E> extends Iterable<E> {
 // Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); // optional
boolean remove(Object element); // optional Iterator<E> iterator();
// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c); // optional
boolean removeAll(Collection<?> c); // optional
boolean retainAll(Collection<?> c); // optional
void clear(); // optional
// Array operations
Object[] toArray();

<T> T[] toArray(T[] a);

}
```

The interface does about what you’d expect given that a Collection represents a group of objects. The interface has methods to tell you how many elements are in the collection (size, isEmpty), to check whether a given object is in the collection (contains), to add and remove an element from the collection (add, remove), and to provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don’t. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

Traversing Collections

There are two ways to traverse collections:

· With the for-each construct and

• By using Iterators

for-each Construct

The for-each construct allows you to concisely traverse a collection or array using a for loop. The following code uses the for-each construct to print out each element of a collection on a separate line:

```
for (Object o : collection)

System.out.println(o);
```

Iterators

An Iterator is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface:

```
public interface Iterator<E> {
boolean hasNext();
E next();
void remove(); // optional
}
```

The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration. The remove method removes the last element that was returned by next from the underlying Collection. The remove method may be called only once per call to next and throws an exception if this rule is violated.

Note that Iterator.remove is the only safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Note: Use Iterator instead of the for-each construct when you need to:

• **Remove the current element**. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.

· **Iterate over multiple collections in parallel.**

The following method shows how to use an Iterator to filter an arbitrary Collection—that is, traverse the collection removing specific elements:

```
static void filter(Collection<?> c) {
for (Iterator<?> it = c.iterator(); it.hasNext(); ) if (!cond(it.next()))
it.remove();
}
```

This simple piece of code is polymorphic, which means that it works for any Collection regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

containsAll Returns true if the target Collection contains all of the elements in the specified Collection.

addAll Adds all of the elements in the specified Collection to the target Collection.

removeAll Removes from the target Collection all of its elements that are also contained in the specified Collection.

retainAll Removes from the target Collection all of its elements that are not also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.

clear Removes all elements from the Collection.

The addAll, removeAll, and retainAll methods all return true if the target Collection was modified in the process of executing the operation.

As a simple example of the power of bulk operations, consider the following idiom to remove all instances of a specified element, e, from a Collection, c:

```
c.removeAll(Collections.singleton(e));
```

More specifically, suppose you want to remove all of the null elements from a Collection:

```
c.removeAll(Collections.singleton(null));
```

This idiom uses Collections.singleton, which is a static factory method that returns an immutable Set containing only the specified element.

Collection Interface Array Operations

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c:

```
Object[] a = c.toArray();
```

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c:

```
String[] a = c.toArray(new String[0]);
```

The Set Interface

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The following is the Set interface:

```
public interface Set<E> extends Collection<E> {
// Basic operations
int size();
boolean isEmpty();
boolean contains(Object element);
boolean add(E element); // optional
boolean remove(Object element); // optional Iterator<E> iterator();
// Bulk operations
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
// optional boolean removeAll(Collection<?> c);
// optional boolean retainAll(Collection<?> c);
 // optional void clear(); // optional
// Array Operations

Object[] toArray();

<T> T[] toArray(T[] a); }
```

The Java platform contains three general-purpose Set implementations: HashSet TreeSet and LinkedHashSet. HashSet, which stores its elements in a hash table, is the best-performing implementation; however, it makes no guarantees concerning the order of iteration. TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Here’s a simple but useful Set idiom. Suppose you have a Collection, c, and you want to create another Collection containing the same elements but with all duplicates eliminated. The following one-liner does the trick:

```
Collection<Type> noDups = new HashSet<Type>(c);
```

It works by creating a Set (which, by definition, cannot contain a duplicate) initially containing all the elements in c. It uses the standard conversion constructor described in The Collection Interface section.

Here is a minor variant of this idiom that preserves the order of the original collection while removing duplicate elements:

```
Collection<Type> noDups = new LinkedHashSet<Type>(c);
```

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed:

```
public static <E> Set<E> removeDups(Collection<E> c) {
return new LinkedHashSet<E>(c);
}
```

Set Interface Basic Operations

The size operation returns the number of elements in the Set (its cardinality). The isEmpty method does exactly what you think it would. The add method adds the specified element to the Set if it’ s not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it’ s present and returns a boolean indicating whether the element was present. The iterator method returns an Iterator over the Set.

The following program takes the words in its argument list and prints out any duplicate words, the number of distinct words, and a list of the words with duplicates eliminated:

```
import java.util.*;
public class FindDups {
public static void main(String[] args) {
Set<String> s = new HashSet<String>();
for (String a : args)
if (!s.add(a))
System.out.println("Duplicate detected: " + a); System.out.println(s.size() + " distinct words: " +
s); }
}
```

Now run the program:

```
java FindDups i came i saw i left
```

The following output is produced:

```
Duplicate detected: i
Duplicate detected: i
4 distinct words: [i, left, saw, came]
```

Note that the code always refers to the Collection by its interface type (Set) rather than by its implementation type (HashSet). This is a strongly recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the Collection's implementation type rather than its interface type, all such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the Set in the preceding example is HashSet, which makes no guarantees as to the order of the elements in the Set. If you want the program to print the word list in alphabetical order, merely change the Set's implementation type from HashSet to TreeSet. Making this trivial one-line change causes the command line in the previous example to generate the following output:

```
java FindDups i came i saw i left
Duplicate detected: i
Duplicate detected: i
4 distinct words: [came, i, left, saw]
```

Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

s1.containsAll(s2) Returns true if s2 is a subset of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)

s1.addAll(s2) Transforms s1 into the union of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)

s1.retainAll(s2) Transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)

s1.removeAll(s2) Transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

To calculate the union, intersection, or set difference of two sets non-destructively (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms:

```
Set<Type> union = new HashSet<Type>(s1); union.addAll(s2);
Set<Type> intersection = new HashSet<Type>(s1); intersection.retainAll(s2);
Set<Type> difference = new HashSet<Type>(s1); difference.removeAll(s2);
```

The implementation type of the result Set in the preceding idioms is HashSet, which is, as already mentioned, the best all-around Set implementation in the Java platform. However, any general-purpose Set implementation could be substituted.

Let's revisit the FindDups program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets—one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets,

which we know how to compute. Here's how the resulting program looks:

```
import java.util.*;
public class FindDups2 {
public static void main(String[] args) {
Set<String> uniques = new HashSet<String>();
Set<String> dups = new HashSet<String>();
 for (String a : args)
if (!uniques.add(a))
dups.add(a);
// Destructive set-difference
uniques.removeAll(dups);
System.out.println("Unique words: " + uniques);
System.out.println("Duplicate words: " + dups); }
}
```

When run with the same argument list used earlier (i came i saw i left), the program yields the following output:

```
 Unique words: [left, saw, came]

Duplicate words: [i]
```

A less common set-algebraic operation is the symmetric set difference—the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively:

```
Set<Type> symmetricDiff = new HashSet<Type>(s1);
symmetricDiff.addAll(s2);
Set<Type> tmp = new HashSet<Type>(s1);
tmp.retainAll(s2));
symmetricDiff.removeAll(tmp);
```

Set Interface Array Operations

The array operations don’t do anything special for Sets beyond what they do for any other Collection. These operations are described in the Collection Interface Array Operations section

The List Interface

A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

Positional access Manipulates elements based on their numerical position in the list.

Search Searches for a specified object in the list and returns its numerical position.

Iteration Extends Iterator semantics to take advantage of the list’s sequential nature.

Range-view Performs arbitrary range operations on the list.

The List interface follows:

```
public interface List<E> extends Collection<E> {
                                        // Positional access
E get(int index);
E set(int index, E element); // optional
boolean add(E element); // optional
void add(int index, E element); // optional
E remove(int index); // optional
boolean addAll(int index,
Collection<? extends E> c); // optional
// Search
int indexOf(Object o);
int lastIndexOf(Object o);
// Iteration
ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);
// Range-view
List<E> subList(int from, int to);
}
```

The Java platform contains two general-purpose List implementations. ArrayList, which is usually the better-performing implementation, and LinkedList which offers better performance under certain circumstances. Also, Vector has been retrofitted to implement List.

Comparison to Vector

If you’ve used Vector, you’re already familiar with the general basics of List. (Of course, List is an interface, while Vector is a concrete implementation.) List fixes several minor API deficiencies in Vector. Commonly used Vector operations, such as elementAt and setElementAt, have been given much shorter names. When you consider that these two operations are the List analog of square brackets for arrays, it becomes apparent that shorter names are highly desirable. Consider the following

assignment statement:

```
a[i] = a[j].times(a[k]);
```

The Vector equivalent is:

```
v.setElementAt(v.elementAt(j).times(v.elementAt(k)), i);
```

The List equivalent is:

```
v.set(i, v.get(j).times(v.get(k)));
```

You may already have noticed that the set method, which replaces the Vector method setElementAt, reverses the order of the arguments so that they match the corresponding array operation. Consider the following assignment statement:

```
gift[5] = "golden rings";
```

The Vector equivalent is:

```
gift.setElementAt("golden rings", 5);
```

The List equivalent is:

```
gift.set(5, "golden rings");
```

For consistency's sake, the method add(int, E), which replaces insertElementAt(Object, int), also reverses the order of the arguments.

The various range operations in Vector (indexOf, lastIndexOf, and setSize) have been replaced by a single range-view operation (subList), which is far more powerful and consistent.

Collection Operations

The operations inherited from Collection all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from Collection, now would be a good time to read the Collection Interface Array Operations section (page 301). The remove operation always removes the first occurrence of the specified element from the list. The add and addAll operations always append the new element(s) to the end of the list. Thus, the following idiom concatenates one list to another:

```
list1.addAll(list2);
```

Here's a nondestructive form of this idiom, which produces a third List consisting of the second list appended to the first:

```
List<Type> list3 = new ArrayList<Type>(list1); list3.addAll(list2);
```

Note that the idiom, in its nondestructive form, takes advantage of ArrayList's standard conversion constructor.

Like the Set interface, List strengthens the requirements on the equals and hashCode methods so that two List objects can be compared for logical equality without regard to their implementation classes. Two List objects are equal if they contain the same elements in the same order.

Positional Access and Search Operations

The basic positional access operations (get, set, add, and remove) behave just like their longer-named counterparts in Vector (elementAt, setElementAt, nsertElementAt, and removeElementAt) with one noteworthy exception: The set and remove perations return the old value that is being overwritten or removed; the Vector counterparts (setElementAt and removeElementAt) return nothing (void). The search operations indexOf and lastIndexOf behave exactly like the identically named operations in Vector.

The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection’s iterator. This call is the positional access analog of Collection’s addAll operation.

Here’s a little method to swap two indexed values in a List.

```
public static <E> void swap(List<E> a, int i, int j) {
    E tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

Of course, there’s one big difference. This is a polymorphic algorithm: It swaps two elements in any List, regardless of its implementation type. Here’s another polymorphic algorithm that uses the preceding swap method:

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i));
}
```

This algorithm, which is included in the Java platform’s Collections class, randomly permutes the specified list using the specified source of randomness. It’s a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it’s fair (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and fast (requiring exactly list.size()-1 swaps). The following program uses this algorithm to print the words in its argument list in random order:

```
import java.util.*;

public class Shuffle {
```

```
public static void main(String[] args) {
List<String> list = new ArrayList<String>();
for (String a : args)
list.add(a);
Collections.shuffle(list, new Random());
System.out.println(list);
}
}
```

In fact, this program can be made even shorter and faster. The Arrays class has a static factory method called asList, which allows an array to be viewed as a List. This method does not copy the array. Changes in the List write through to the array and vice versa. The resulting List is not a general-purpose List implementation, because it doesn’t implement the (optional) add and remove operations: Arrays are not resizable. Taking advantage of Arrays.asList and calling the library version of shuffle, which uses a default source of randomness, you get the following tiny program whose behavior is identical to the previous program:

```
import java.util.*;
public class Shuffle {
public static void main(String[] args) {
List<String> list = Arrays.asList(args);
Collections.shuffle(list);
System.out.println(list);
}
}
```

Iterators

As you’d expect, the Iterator returned by List’s iterator operation returns the elements of the list in proper sequence. List also provides a richer iterator, called a ListIterator, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator. The ListIterator interface follows:

```
 public interface ListIterator<E> extends Iterator<E> { boolean hasNext();
E next();
boolean hasPrevious();
E previous();
int nextIndex();
int previousIndex();
void remove(); // optional
void set(E e); // optional
void add(E e); // optional
}
```

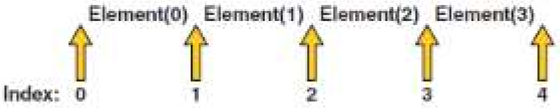
The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previous operations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

Here's the standard idiom for iterating backward through a list:

```
for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {

Type t = it.previous();

...

}
```

Note the argument to listIterator in the preceding idiom. The List interface has two forms of the listIterator method. The form with no arguments returns a ListIterator positioned at the beginning of the list; the form with an int argument returns a ListIterator positioned at the specified index. The index refers to the element that would be returned by an initial call to next. An initial call to previous would return the element whose index was index-1. In a list of length n, there are n+1 valid values for index, from 0 to n, inclusive.

Intuitively speaking, the cursor is always between two elements—the one that would be returned by a call to previous and the one that would be returned by a call to next. The n+1 valid index values correspond to the n+1 gaps between elements, from the gap before the first element to the gap after the last one. Following Figure shows the five possible cursor positions in a list containing four elements.



Calls to next and previous can be intermixed, but you have to be a bit careful. The first call to previous returns the same element as the last call to next. Similarly, the first call to next after a sequence of calls to previous returns the same element as the last call to previous.

It should come as no surprise that the nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous. These calls are typically used either to report the position where something was found or to record the position of the ListIterator so that another ListIterator with identical position can be created.

It should also come as no surprise that the number returned by nextIndex is always one greater than the number returned by previousIndex. This implies the behavior of the two boundary cases: (1) a call to previousIndex when the cursor is before the initial element returns -1 and (2) a call to nextIndex when the cursor is after the final element returns list.size(). To make all this concrete, the following is a possible implementation of List.indexOf:

```
 public int indexOf(E e) {
for (ListIterator<E> it = listIterator(); it.hasNext(); )
 if (e == null ? it.next() == null : e.equals(it.next()))
return it.previousIndex();
return -1; // Element not found
}
```

Note that the indexOf method returns it.previousIndex() even though it is traversing the list in the forward direction. The reason is that it.nextIndex() would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The Iterator interface provides the remove operation to remove the last element returned by next from the Collection. For ListIterator, this operation removes the last element returned by next or previous. The ListIterator interface provides two additional operations to modify the list—set and add. The set method overwrites the last element returned by next or previous with the specified element. The following polymorphic algorithm uses set to replace all occurrences of one specified value with another:

```
 public static <E> void replace(List<E> list, E val, E newVal) {
 for (ListIterator<E> it = list.listIterator(); it.hasNext();) if (val == null ?
it.next() == null : val.equals(it.next()))
it.set(newVal);
}
```

The only bit of trickiness in this example is the equality test between val and it.next. You need to special-case a val value of null to prevent a NullPointerException.

The add method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list:

```
 public static <E> void replace(List<E> list, E val, List<? extends E>
newVals) {
for (ListIterator<E>
it = list.listIterator(); it.hasNext(); ){
if (val == null ?
it.next() == null : val.equals(it.next())) {
it.remove();
for (E e : newVals)

it.add(e);
}

}

}
```

Range-View Operation

The range-view operation, subList(int fromIndex, int toIndex), returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. This half-open range mirrors the typical for loop:

```
for (int i = fromIndex; i < toIndex; i++) {

...

}
```

As the term view implies, the returned List is backed up by the List on which subList was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by passing a subList view instead of a whole List. For example, the following idiom removes a range of elements from a List:

```
list.subList(fromIndex, toIndex).clear();
```

Similar idioms can be constructed to search for an element in a range:

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

Note that the preceding idioms return the index of the found element in the subList, not the index in the backing List.

Any polymorphic algorithm that operates on a List, such as the replace and shuffle examples, works with the List returned by subList.

Here’s a polymorphic algorithm whose implementation uses subList to deal a hand from a deck. That is, it returns a new List (the “hand”) containing the specified number of elements taken from the end of the specified List (the “deck”). The elements returned in the hand are removed from the deck:

```
public static <E> List<E> dealHand(List<E> deck, int n) {

int deckSize = deck.size();
List<E> handView = deck.subList(deckSize - n, deckSize);
 List<E> hand = new ArrayList<E>(handView);
 handView.clear();
return hand;
}
```

Note that this algorithm removes the hand from the end of the deck. For many common List implementations, such as ArrayList, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is a program that uses the dealHand method in combination with Collections.shuffle to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand:

```
 import java.util.*;
class Deal {
public static void main(String[] args) {
int numHands = Integer.parseInt(args[0]);
int cardsPerHand = Integer.parseInt(args[1]);
// Make a normal 52-card deck.
String[] suit = new String[]
{"spades", "hearts", "diamonds", "clubs"};
String[] rank = new String[]
{"ace","2","3","4","5","6","7","8",
"9","10","jack","queen","king"};
List<String> deck = new ArrayList<String>();
 for (int i = 0; i < suit.length; i++)
for (int j = 0; j < rank.length; j++)
deck.add(rank[j] + " of " + suit[i]);
Collections.shuffle(deck);
for (int i=0; i < numHands; i++)

 System.out.println(dealHand(deck, cardsPerHand)); }

}
```

Running the program produces the following output:

```
% java Deal 4 5


[8 of hearts, jack of spades, 3 of spades, 4 of spades, king of diamonds]
```

```
 [4 of diamonds, ace of clubs, 6 of clubs, jack of hearts, queen of hearts]
[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds, 9 of clubs]
 [8 of spades, 6 of diamonds, ace of spades, 3 of hearts, ace of hearts]
```

Although the subList operation is extremely powerful, some care must be exercised when using it. The semantics of the List returned by subList become undefined if elements are added to or removed from the backing List in any way other than via the returned List. Thus, it's highly recommended that you use the List returned by subList only as a transient object—to perform one or a sequence of range operations on the backing List. The longer you use the subList instance, the greater the probability that you'll compromise it by modifying the backing List directly or through another subList object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

List Algorithms

Most polymorphic algorithms in the Collections class apply specifically to List. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms

sort Sorts a List using a merge sort algorithm, which provides a fast, stable sort.
(A stable sort is one that does not reorder equal elements.)
shuffle Randomly permutes the elements in a List.
reverse Reverses the order of the elements in a List.
rotate Rotates all the elements in a List by a specified distance.
swap Swaps the elements at specified positions in a List.
replaceAll Replaces all occurrences of one specified value with another.
fill Overwrites every element in a List with the specified value.
copy Copies the source List into the destination List.
binarySearch Searches for an element in an ordered List using the binary search algorithm.

indexOfSubList Returns the index of the first sublist of one List that is equal to another.

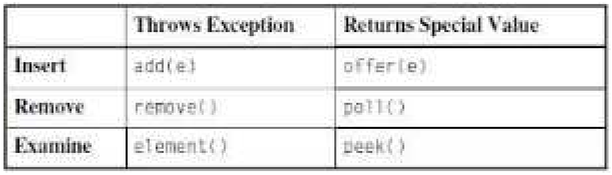
lastIndexOfSubList Returns the index of the last sublist of one List that is equal to another.

The Queue Interface

A Queue is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface follows:

```
 public interface Queue<E> extends Collection<E> { E element();
boolean offer(E e);
E peek();
E poll();
E remove();
}
```

Each Queue method exists in two forms: (1) one throws an exception if the operation fails, and (2) the other returns a special value if the operation fails (either null or false, depending on the operation).



	Throws Exception	Returns Special Value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

Queues typically, but not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to their values—see the Object Ordering section (page 328) for details. Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

It is possible for a Queue implementation to restrict the number of elements that it holds; such queues are known as bounded. Some Queue implementations in java.util.concurrent are bounded, but the implementations in java.util are not.

The add method, which Queue inherits from Collection, inserts an element unless it would violate the queue’s capacity restrictions, in which case it throws IllegalStateException. The offer method, which is intended solely for use on bounded queues, differs from add only in that it indicates failure to insert an element by returning false.

The remove and poll methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue’s ordering policy. The remove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, while poll returns null.

The element and peek methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as remove and poll: If the queue is empty, element throws NoSuchElementException, while peek returns null.

Queue implementations generally do not allow insertion of null elements. The LinkedList implementation, which was retrofitted to implement Queue, is an exception. For historical reasons, it permits null elements, but you should refrain from taking advantage of this, because null is used as a special return value by the poll and peek methods.

Queue implementations generally do not define element-based versions of the equals and hashCode methods but instead inherit the identity-based versions from Object.

The Queue interface does not define the blocking queue methods, which are common in concurrent programming. These methods, which wait for elements to appear or for space to become available, are defined in the interface java.util.concurrent.BlockingQueue, which extends Queue.

In the following example program, a queue is used to implement a countdown timer. The queue is preloaded with all the integer values from a number specified on the command line to zero, in descending order. Then, the values are removed from the queue and printed at one-second intervals. The program is artificial in that it would be more natural to do the same thing without using a queue, but it illustrates the use of a queue to store elements prior to subsequent processing:

```
import java.util.*;
public class Countdown {
public static void main(String[] args)
throws InterruptedException {
int time = Integer.parseInt(args[0]);
Queue<Integer> queue = new LinkedList<Integer>();
for (int i = time; i >= 0; i--)
queue.add(i);
while (!queue.isEmpty()) {
System.out.println(queue.remove());
Thread.sleep(1000);
}
}
}
```

In the following example, a priority queue is used to sort a collection of elements.
Again this program is artificial in that there is no reason to use it in favor of the sort method provided in Collections, but it illustrates the behavior of priority queues:

```
static <E> List<E> heapSort(Collection<E> c) {
 Queue<E> queue = new PriorityQueue<E>(c);
 List<E> result = new ArrayList<E>();
while (!queue.isEmpty())
result.add(queue.remove());
return result;
}
```

The Map Interface

A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical function abstraction.

The Map interface follows:

```
public interface Map<K,V> {
// Basic operations
V put(K key, V value);

 V get(Object key);
V remove(Object key);
boolean containsKey(Object key);
boolean containsValue(Object value);
int size();
boolean isEmpty();
// Bulk operations
void putAll(Map<? extends K, ? extends V> m); void clear();
// Collection Views
public Set<K> keySet();
public Collection<V> values();
public Set<Map.Entry<K,V>> entrySet();
// Interface for entrySet elements
public interface Entry {
K getKey();
V getValue();
V setValue(V value);
}
}
```

The Java platform contains three general-purpose Map implementations: HashMap, TreeMap, and LinkedHashMap. Their behavior and performance are precisely analogous to HashSet, TreeSet, and LinkedHashSet. Also, Hashtable was retrofitted to implement Map.

Comparison to Hashtable

If you've used Hashtable, you're already familiar with the general basics of Map. (Of course, Map is an interface, while Hashtable is a concrete implementation.) The following are the major differences:

- Map provides Collection views instead of direct support for iteration via Enumeration objects. Collection views greatly enhance the expressiveness of the interface, as discussed later in this section.

- Map allows you to iterate over keys, values, or key-value pairs; Hashtable does not provide the third option.
- Map provides a safe way to remove entries in the midst of iteration; Hashtable did not.

Finally, Map fixes a minor deficiency in the Hashtable interface. Hashtable has a method called contains, which returns true if the Hashtable contains a given value. Given its name, you'd expect this method to return true if the Hashtable contained a given key, because the key is the primary access mechanism for a Hashtable. The Map interface eliminates this source of confusion by renaming the method containsValue. Also, this improves the interface's consistency—containsValue parallels containsKey.

Map Interface Basic Operations

The basic operations of Map (put, get, containsKey, containsValue, size, and isEmpty) behave exactly like their counterparts in Hashtable. The following program generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list:

```
import java.util.*;
public class Freq {
public static void main(String[] args) {
Map<String, Integer> m = new HashMap<String,
 Integer>();
// Initialize frequency table from command line for (String a : args) {
Integer freq = m.get(a);

m.put(a, (freq == null) ? 1 : freq + 1);

}

System.out.println(m.size() + " distinct words:"); System.out.println(m);

}

}
```

The only tricky thing about this program is the second argument of the put statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

```
java Freq if it is to be it is up to me to delegate
```

The program yields the following output:

```
8 distinct words:

{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}
```

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the Map from HashMap to TreeMap. Making this four-character change causes the program to generate the following output from the same command line:

```
 8 distinct words:

{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}
```

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map to LinkedHashMap. Doing so results in the following output:

```
8 distinct words:

{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}
```

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the Set and List interfaces, Map strengthens the requirements on the equals and hashCode methods so that two Map objects can be compared for logical equality without regard to their implementation types. Two Map instances are equal if they represent the same key-value mappings.

By convention, all general-purpose Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map. This standard Map conversion constructor is entirely analogous to the standard Collection constructor: It allows the caller to create a Map of a desired implementation type that initially contains all of the mappings in another Map, regardless of the other Map's implementation type. For example, suppose you have a Map, named m. The following one-liner creates a new HashMap initially containing all of the same key-value mappings as m:

```
 Map<K, V> copy = new HashMap<K, V>(m);
```

Map Interface Bulk Operations

The clear operation does exactly what you would think it could do: It removes all the mappings from the Map. The putAll operation is the Map analogue of the Collection interface' s addAll operation. In addition to its obvious use of dumping one Map into another, it has a second, more subtle use. Suppose a Map is used to represent a collection of attribute-value pairs; the putAll operation, in combination with the Map conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique:

```
 static <K, V> Map<K, V> newAttributeMap(
Map<K, V>defaults, Map<K, V> overrides) {
 Map<K, V> result = new HashMap<K, V>(defaults);
result.putAll(overrides);
return result;
}
```

Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:
keySet The Set of keys contained in the Map.

values The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.

entrySet The Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.

The Collection views provide the only means to iterate over a Map. This example illustrates the standard idiom for iterating over the keys in a Map with a for-each construct:

```
for (KeyType key : m.keySet())

System.out.println(key);
```

and with an iterator:

```
// Filter a map based on some property of its keys.

for (Iterator<Type> it = m.keySet().iterator();

it.hasNext(); ) if (it.next().isBogus())

it.remove()

;
```

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs:

```
 for (Map.Entry<KeyType, ValType> e : m.entrySet()) System.out.println(e.getKey() + ": " +
e.getValue());
```

At first, many people worry that these idioms may be slow because the Map has to create a new Collection instance each time a Collection view operation is called. Rest easy: There’s no reason that a Map cannot always return the same object each time it is asked for a given Collection view. This is precisely what all the Map implementations in java.util do.

With all three Collection views, calling an Iterator’s remove operation removes the associated entry from the backing Map, assuming that the backing Map supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry’s setValue method during iteration (again, assuming the Map supports value modification to begin with). Note that these are the only safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

The Collection views support element removal in all its many forms—remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation. (Yet again, this assumes that the backing Map supports element removal.)

The Collection views do not support element addition under any circumstances. It would make no sense for the keySet and values views, and it’s unnecessary for the entrySet view, because the backing Map’s put and putAll methods provide the same functionality.

12

JAVA GUI Development with Swing

12.1 Introduction

Swing toolkit includes a rich set of components for building GUIs and adding interactivity to Java applications. Swing includes all the components you would expect from a modern toolkit: table controls, list controls, tree controls, buttons, and labels.

Swing is far from a simple component toolkit, however. It includes rich undo support, a highly customizable text package, integrated internationalization and accessibility support. To truly leverage the cross-platform capabilities of the Java platform, Swing supports numerous look and feels, including the ability to create your own look and feel. The ability to create a custom look and feel is made easier with Synth, a look and feel specifically designed to be customized. Swing wouldn't be a component toolkit without the basic user interface primitives such as drag and drop, event handling, customizable painting, and window management.

Swing is part of the Java Foundation Classes (JFC). The JFC also include other features important to a GUI program, such as the ability to add rich graphics functionality and the ability to create a program that can work in different languages and by users with different input devices.

The following list shows some of the features that Swing and the Java Foundation Classes provide.

Swing GUI Components (Lightweight JAVA GUI Components)

The Swing toolkit includes a rich array of components: from basic components, such as buttons and check boxes, to rich and complex components, such as tables and text. Even deceptively simple components, such as text fields, offer sophisticated functionality, such as formatted text input or password field behavior. There are file browsers and dialogs to suit most needs, and if not, customization is possible. If none of Swing's provided components are exactly what you need, you can leverage the basic Swing component functionality to create your own.

Java 2D API

To make your application stand out; convey information visually; or add figures, images, or animation to your GUI, you'll want to use the Java 2DTM API. Because Swing is built on the 2D package, it's trivial to make use of 2D within Swing components. Adding images, drop shadows, compositing — it's easy with Java 2D.

Pluggable Look-and-Feel Support

Any program that uses Swing components has a choice of look and feel. The JFC classes shipped by Sun and Apple provide a look and feel that matches that of the platform. The Synth package allows you to create your own look and feel. The GTK+ look and feel makes hundreds of existing look and feels available to Swing programs. A program can specify the look and feel of the platform it is running on, or it can specify to always use the Java look and feel, and without recompiling, it will just work. Or, you can ignore the issue and let the UI manager sort it out.

Data Transfer

Data transfer, via cut, copy, paste, and drag and drop, is essential to almost any application. Support for data transfer is built into Swing and works between Swing components within an application, between Java applications, and between Java and native applications.

Internationalization

This feature allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. Applications can be created that accept input in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

Swing's layout managers make it easy to honor a particular orientation required by the UI. For example, the UI will appear right to left in a locale where the text flows right to left. This support is automatic: You need only code the UI once and then it will work for left to right and right to left, as well as honor the appropriate size of components that change as you localize the text.

Accessibility API

People with disabilities use special software — assistive technologies — that mediates the user experience for them. Such software needs to obtain a wealth of information about the running application in order to represent it in alternate media: for a screen reader to read the screen with synthetic speech or render it via a Braille display, for a screen magnifier to track the caret and keyboard focus, for on-screen keyboards to present dynamic keyboards of the menu choices and toolbar items and dialog controls, and for voice control systems to know what the user can control with his or her voice. The accessibility API enables these assistive technologies to get the information they need, and to programmatically manipulate the elements that make up the graphical user interface.

Undo Framework API

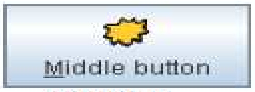
Swing's undo framework allows developers to provide support for undo and redo. Undo support is built in to Swing's text component. For other components, Swing supports an unlimited number of actions to undo and redo, and is easily adapted to an application. For example, you could easily enable undo to add and remove elements from a table.

Flexible Deployment Support

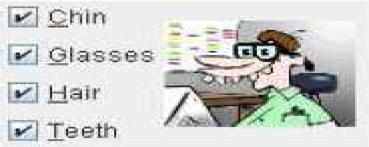
If you want your program to run within a browser window, you can create it as an applet and run it using Java Plug-in, which supports a variety of browsers, such as Internet Explorer, Firefox, and Safari. If you want to create a program that can be launched from a browser, you can do this with Java Web Start. Of course, your application can also run outside of browser as a standard desktop application.

12.2 Swing GUI Components

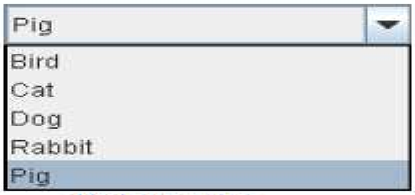
Basic controls : simple components that are primarily used to get input from the user; they may also show simple state.



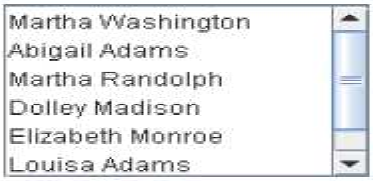
JButton



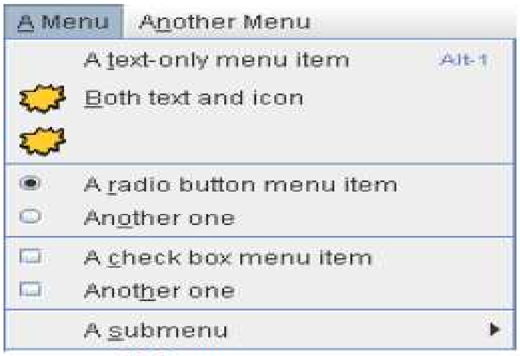
JCheckBox



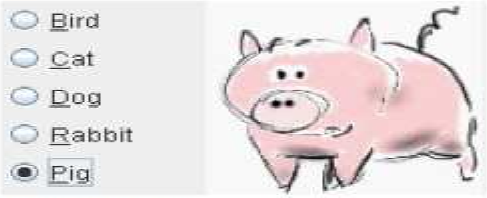
JComboBox



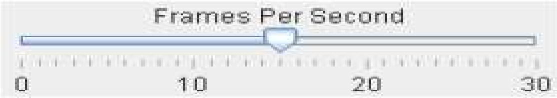
JList



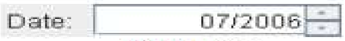
JMenu



JRadioButton

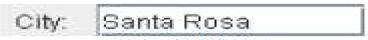


JSlider



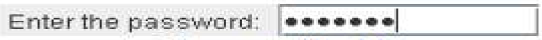
Date: 07/2006

JSpinner



City: Santa Rosa

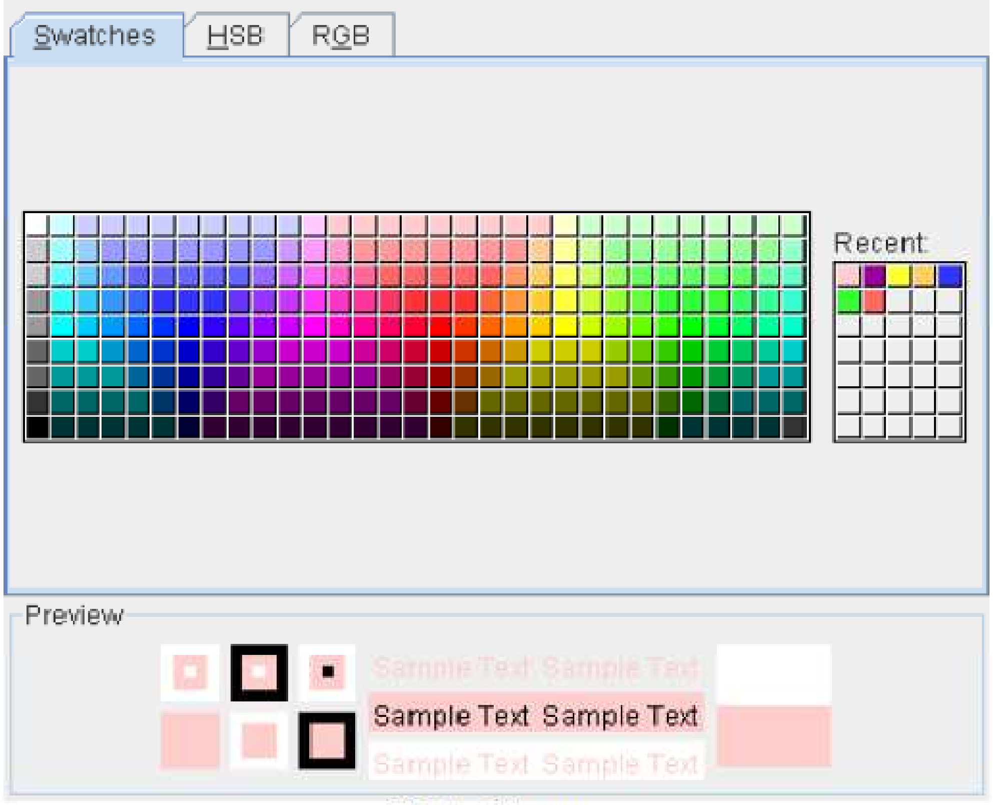
JTextField



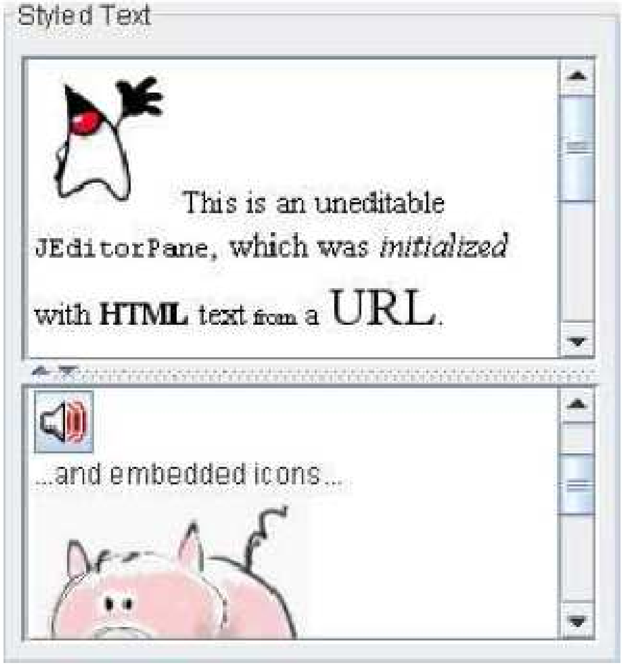
Enter the password: ●●●●●●●

JPasswordField

Interactive Displays of Highly Formatted Information: These components display highly formatted information that (if you choose) can be modified by the user.



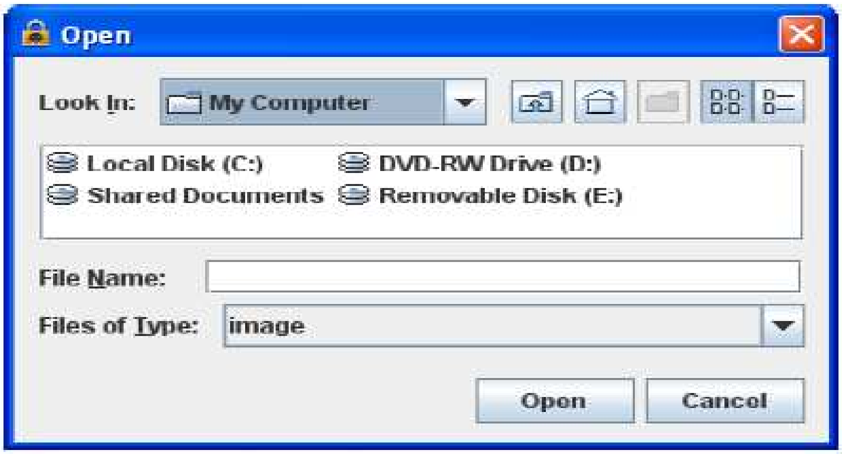
JColorChooser



This is an uneditable `JEditorPane`, which was *initialized* with **HTML** text from a URL.

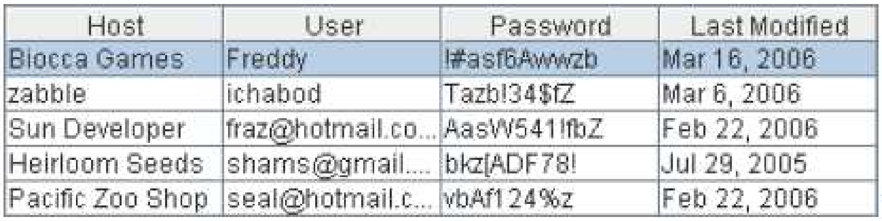
...and embedded icons...

JEditorPane & JTextPane



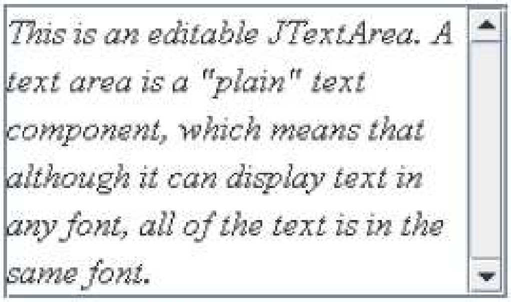
Files of Type: image

JFileChooser



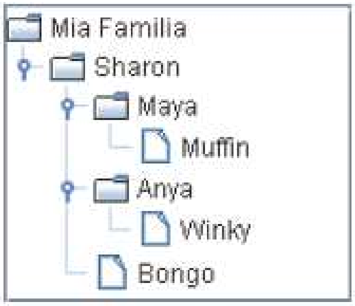
Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

JTable



This is an editable JTextArea. A text area is a "plain" text component, which means that although it can display text in any font, all of the text is in the same font.

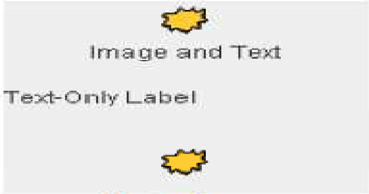
JTextArea



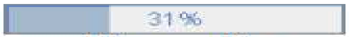
JTree

Uneditable Information Displays

These components exist solely to give the user information.

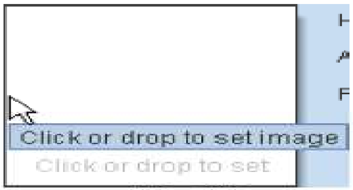


JLabel



JProgressBar

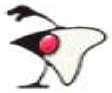
JSeparator



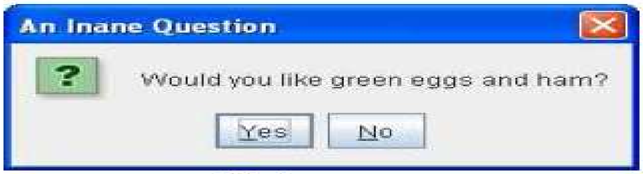
JToolTip

Top-Level Containers

At least one of these components must be present in any Swing application.

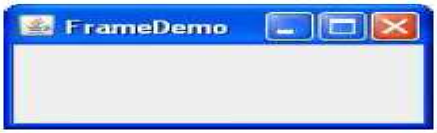


JApplet



Would you like green eggs and ham?

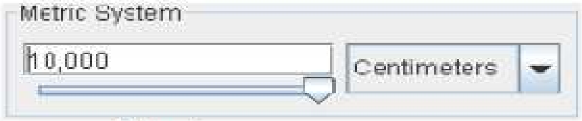
JDialog



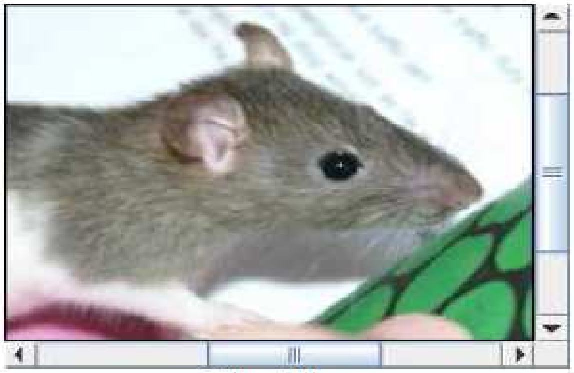
JFrame

General-Purpose Containers

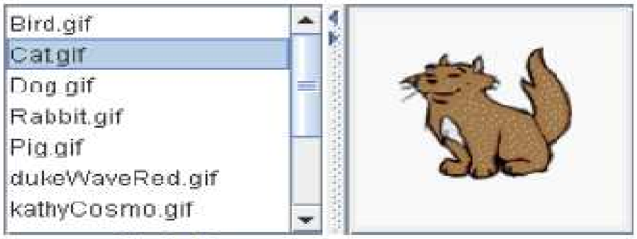
These general-purpose containers are used in most Swing applications.



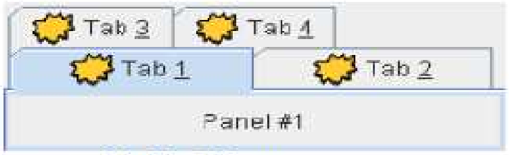
JPanel



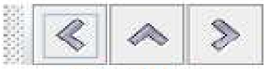
JScrollPane



JSplitPane



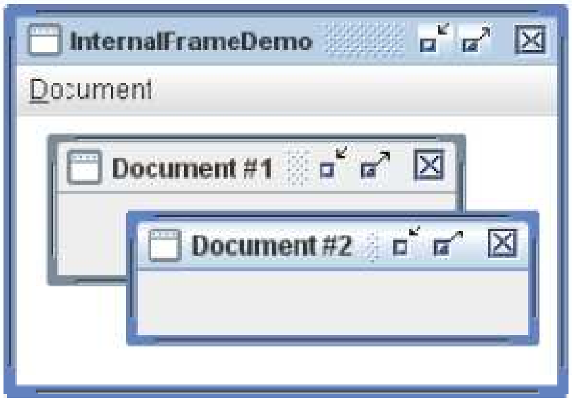
JTabbedPane



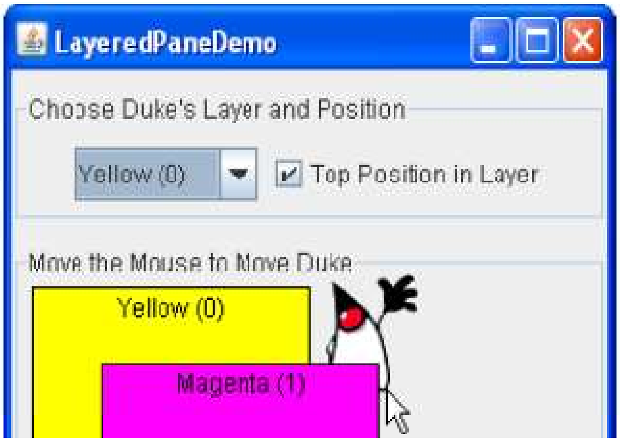
JToolBar

Special-Purpose Containers

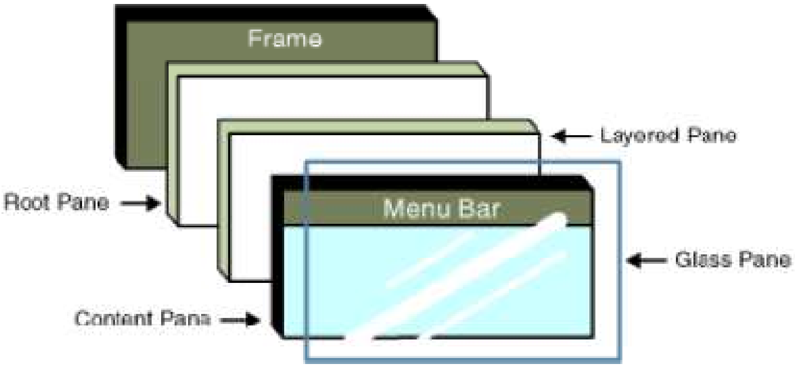
These special-purpose containers play specific roles in the UI



JInternalFrame



JLayeredPane



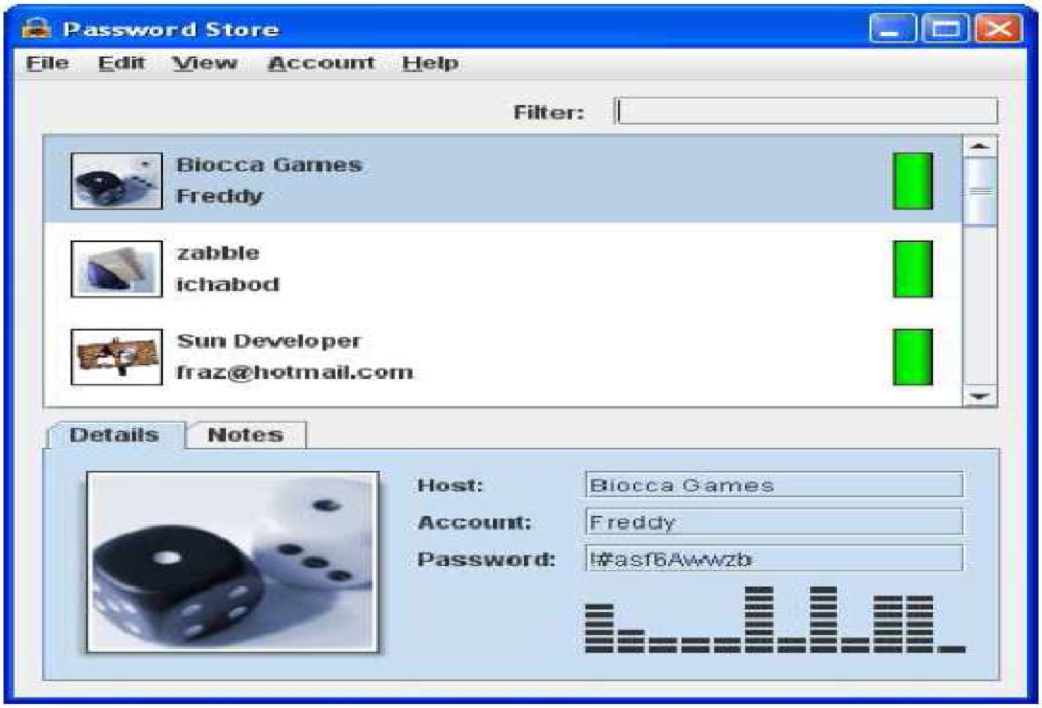
RootPane

12.3 Pluggable Look and Feel

The Swing toolkit allows you to decide how to configure the particular look and feel of your application. If you don't specify a look and feel, the Swing UI manager figures out which one to use. The options for setting a look and feel include:

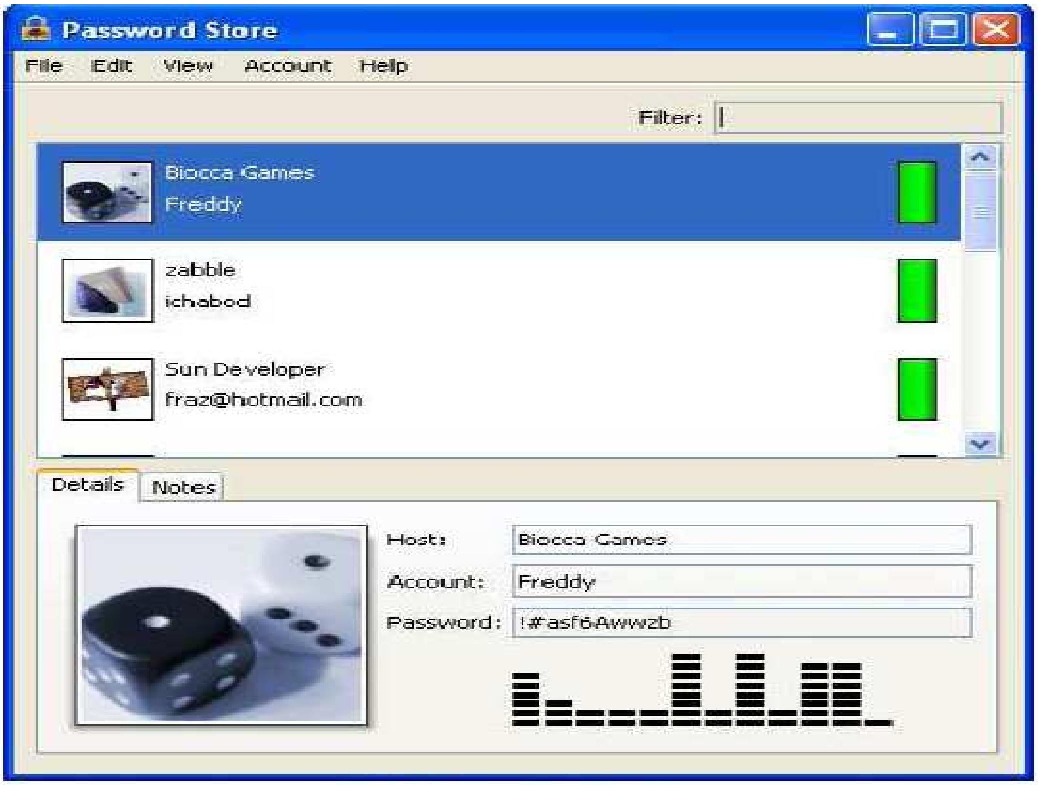
- Leave it up to the Swing UI manager. If a particular look and feel is not specified by the program, Swing's UI manager checks whether the user has specified a preference. If that preference hasn't been specified or isn't available, the default look and feel is used. The default look and feel is determined by the supplier of the JRE. For the JRE that Sun provides, the Java look and feel (called Metal) is used. The Java look and feel works on all platforms.
- Use the look and feel of the native platform. If the application is running on a Microsoft Windows XP machine, the Windows look and feel is used. On Mac OS platforms, the Aqua look and feel is used. On UNIX platforms, such as Solaris or Linux, either the GTK+ look and feel or the CDE/Motif look and feel is used, depending on the user's desktop choice.
- Specify a particular look and feel. Swing ships with four look and feels: Java (also called Metal), Microsoft Windows, GTK+, and CDE/Motif. The GTK+ look and feel requires a theme, and there are many available for free on the Internet.

- Create your own look and feel using the Synth package.
- Use an externally provided look and feel.



Host: Biocca Games
Account: Freddy
Password: !#asf6Awwzb

JAVA Look & Feel



Host: Biocca Games
Account: Freddy
Password: !#asf6Awwzb

Windows Look & Feel

12.4 Data Transfer

The Swing toolkit supports the ability to transfer data between components within the same Java application, between different Java applications, and between Java and native applications. Data can be transferred via a drag and drop gesture, or via the clipboard using cut, copy, and paste.

Drag and Drop

Drag-and-drop support can be easily enabled for many of Swing's components (sometimes with a single line of code). For example, it's trivial to enable drag and drop and copy and paste support for JTable, Swing's table component. All you need to provide is the data representing the selection and how to get your data from the clipboard – that's it!

Cut, Copy, and Paste

Most of the text-based components, such as editor pane and text field, support cut/copy and paste out of the box. Of course, menu items need to be created and "wired up" to the appropriate actions. Other components, such as list and tree, can support cut, copy, and paste with some minimal work.

12.5 Internationalization and Localization

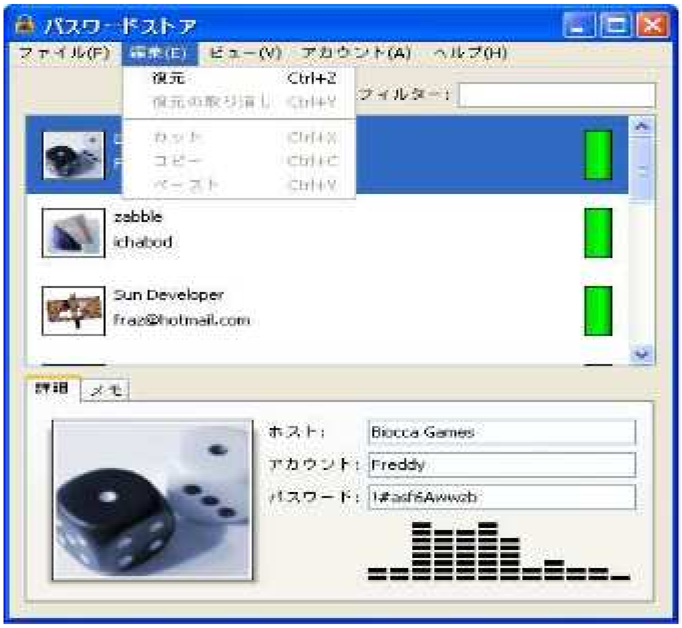
Internationalization is the process of designing an application so that the user can run it using his or her cultural preferences without modifying or recompiling the code. These cultural preferences, collectively known as locale, include (but aren't limited to): language, currency formatting, time and date formatting, and numeric formatting. An internationalized program is designed so that text elements, such as status messages and GUI component labels, are stored outside the source code in resource bundles and retrieved dynamically. Separating the locale-specific information from the code is what allows a program to run in different languages and with different preferences without having to recompile.

Localization is the process of translating the text to a particular language and adding any locale-specific components. When an application is localized to a language and you run the app in that locale, Swing grabs the localized strings from the resource bundle and the layout manager resizes the component accordingly. For example, an English-speaking person writes an application following the rules of internationalization; later, that application is localized to Japanese and Spanish. When a user with the Language System Preference set to Japanese runs the application, Swing detects this. When the application appears, the menus, labels, buttons, and so on, show Japanese text, and the components are scaled accordingly. If that user then quits the program, sets the language system preference to Spanish, and re-launches the application, the application appears in Spanish, scaled according to the new character set.

Swing's layout managers understand how locale affects a UI — it is not necessary to create a new layout for each locale. For example, in a locale where text flows right to left, the layout manager will arrange components in the same orientation, if specified. Bidi text (mixed directional text, used by Hebrew and Arabic, for example) is supported as well.

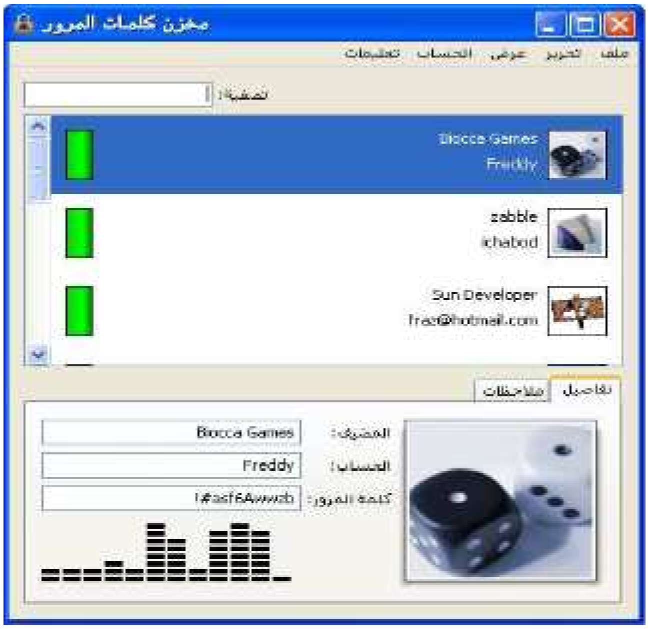
Every program should be designed with internationalization in mind: GUI component labels, status messages, currency, date, phone, and address formats should not be hardcoded into programs. Once a program has been internationalized, a language expert can perform the actual translation at a later date without requiring any recompiling.

As the following screenshots show, the application has been localized to Japanese and Arabic.



ホスト: Biocca Games
アカウント: Freddy
パスワード: 1#asf6Awwzb

Using Japanese Locale



Blocca Games
Freddy

zabble
ichabod

Sun Developer
fraz@hotmail.com

المضيف: Blocca Games
الحساب: Freddy
كلمة المرور: !#asf6Awwzb

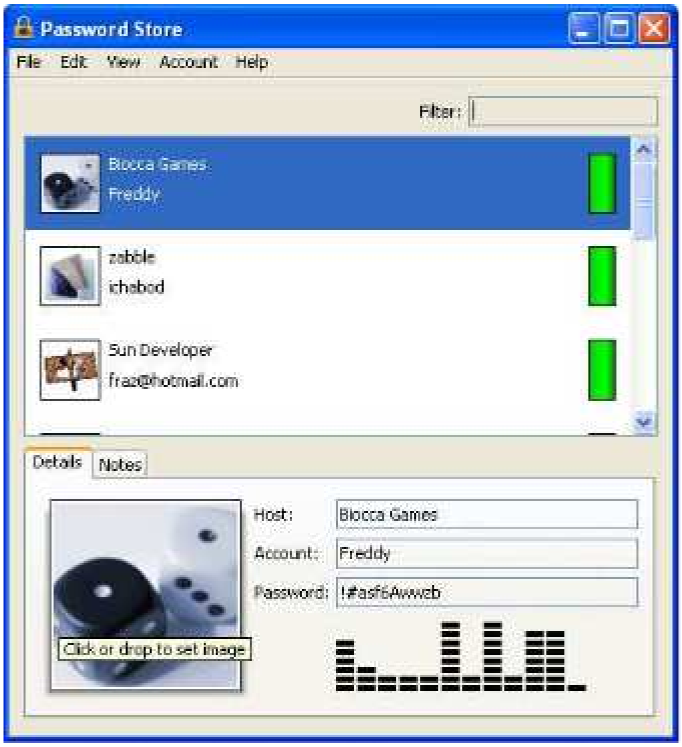
Using Arabic Locale

12.6 Accessibility

Assistive technologies exist to enable people with permanent or temporary disabilities to use the computer. This includes a wide variety of techniques and equipment — voice interfaces, magnifiers, screen readers, closed captioning, keyboard enhancements, and so on. In many countries, including the United States, Australia, Canada, and the European Union, there are laws requiring that programs function smoothly with assistive technologies.

A certain level of accessibility is built-in to all Swing components, but full accessibility can be achieved by following some simple rules. For example, assign tool tips, keyboard alternatives, and textual descriptions for images, wherever possible.

The PasswordStore demo follows the rules set out for accessibility. In the following figure, you can see an example of tool tip text.



Blocca Games
Freddy

zabble
ichabod

Sun Developer
fraz@hotmail.com

Host: Blocca Games
Account: Freddy
Password: !#asf6Awweb

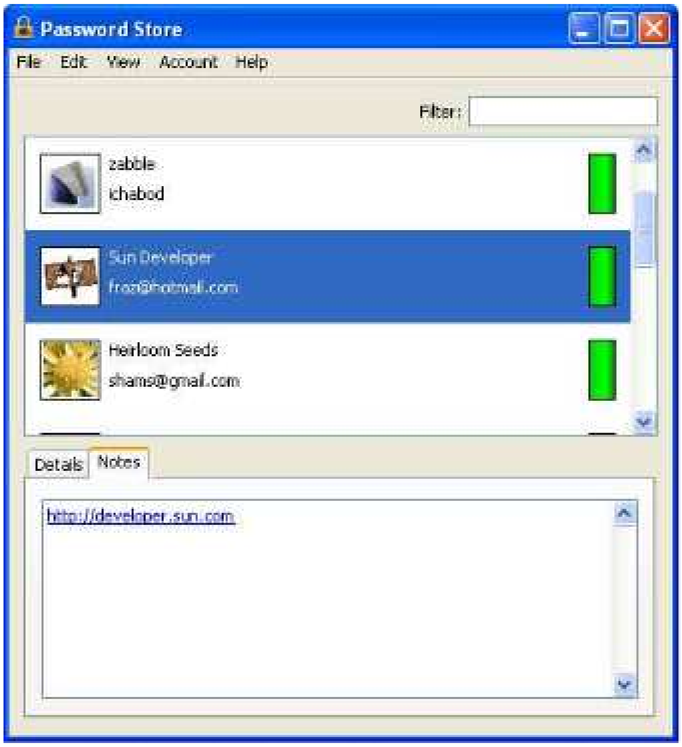
PasswordStore With a Tooltip

12.7 Integrating with the Desktop

The Desktop API, introduced in version 6 of the Java Platform, Standard Edition (Java SE), enables Java applications to integrate seamlessly with the desktop. Three types of integration are supported:

• The ability to launch the host system's default browser with a specific Uniform Resource Identifier (URI).
• The ability to launch the host system's default email client.
· The ability to launch applications to open, edit, or print files associated with those applications.

You can see this in the PasswordStore demo in the Notes text pane. Click on the link that is displayed in the text pane — it opens the specified URI in the default browser.



zabble
ichabod

Sun Developer
frozi@hotmail.com

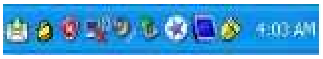
Heirloom Seeds
shams@gmail.com

http://developer.sun.com

Click on the URI and it opens in the Default Browser

12.8 System Tray Icon Support

The desktop of some platforms, such as Microsoft Windows, includes a system tray, as shown in the following screenshot:



On Microsoft Windows, it is called the "Taskbar Status Area." On Gnome, the "Notification Area", and on KDE, the "System Tray." However it may be called, the system tray is shared by all applications.

On platforms where it is supported, an application may insert a mini-icon, called a Tray Icon, into the system tray. This icon can be used to notify the user of a change in the application's status, or a need to take a particular action. Clicking the tray icon can bring up the application window. A popup menu and a tooltip can also be attached to the tray icon.

System Tray support has been added in version 6 of Java SE.

13

SCJP Digest

Java Programming Language Keywords

1. Keywords cannot be used as identifiers (names) for classes, methods, variables, or anything else in your code.

2. All keywords start with a lowercase letter.

Literals and Ranges of All Primitive Data Types

1. All six number types in Java are signed, so they can be positive or negative.
2. Use the formula -2(bits-1) to 2(bits-1)-1 to determine the range of an integer type.
3. A char is really a 16-bit unsigned integer.
4. Literals are source code representations of primitive data types, or String.
5. Integers can be represented in octal (0127), decimal (1245), and hexadecimal 0XCAFE).
6. Numeric literals cannot contain a comma.
7. A char literal can be represented as a single character in single quotes (‘A’).
8. A char literal can also be represented as a Unicode value (‘\u0041’).
9. A char literal can also be represented as an integer, as long as the integer is less than 65536.
10. A boolean literal can be either true or false.
11. Floating-point literals are always double by default; if you want a float, you must append an F or f to the literal.

Array Declaration, Construction, and Initialization

1. Arrays can hold primitives or objects, but the array itself is always an object.
2. When you declare an array, the brackets can be to the left or right of the variable name.
3. It is never legal to include the size of an array in the declaration.
4. You must include the size of an array when you construct it (using new) unless you are creating an anonymous array.
5. Elements in an array of objects are not automatically created, although primitive array elements are given default values.
6. You’ll get a NullPointerException if you try to use an array element in an object array, if that element does not refer to a real object.
7. Arrays are indexed beginning with zero. In an array with three elements, you can access element 0,element 1, and element 2.
8. You’ll get an ArrayIndexOutOfBoundsException if you try to access outside the range of an array.
9. Arrays have a length variable that contains the number of elements in the array.
10. The last index you can access is always one less than the length of the array.
11. Multidimensional arrays are just arrays of arrays.
12. The dimensions in a multidimensional array can have different lengths.
13. An array of primitives can accept any value that can be promoted implicitly
14. to the declared type of the array. For example, a byte variable can be placed in an int array.

15. An array of objects can hold any object that passes the IS-A (or instanceof) test for the declared type of the array. For example, if Horse extends Animal, then a Horse object can go into an Animal array.
16. If you assign an array to a previously declared array reference, the array you're assigning must be the same dimension as the reference you're assigning it to.

17. You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a Honda array can be assigned to an array declared as type Car (assuming Honda extends Car).

Using a Variable or Array Element That Is Uninitialized and Unassigned

1. When an array of objects is instantiated, objects within the array are not instantiated automatically, but all the references get the default value of null.
2. When an array of primitives is instantiated, all elements get their default values.
3. Just as with array elements, instance variables are always initialized witha default value.
4. Local/automatic/method variables are never given a default value. If you attempt to use one before initializing it, you' ll get a compiler error.

Command-Line Arguments to Main

1. Command-line arguments are passed to the String array parameter in the main method.
2. The first command-line argument is the first element in the main String array parameter.
3. If no arguments are passed to main, the length of the main String array parameter will be zero.

Class Access Modifiers

1. There are three access modifiers: public, protected, and private.
2. There are four access levels: public, protected, default, and private.
3. Classes can have only public or default access.
4. Class visibility revolves around whether code in one class can:
5. Create an instance of another class
6. Extend (or subclass), another class
7. Access methods and variables of another class
8. A class with default access can be seen only by classes within the same package.
9. A class with public access can be seen by all classes from all packages.

Class Modifiers (nonaccess)

1. Classes can also be modified with final, abstract, or strictfp.
2. A class cannot be both final and abstract.
3. A final class cannot be subclassed.
4. An abstract class cannot be instantiated.
5. A single abstract method in a class means the whole class must be abstract.
6. An abstract class can have both abstract and nonabstract methods.
7. The first concrete class to extend an abstract class must implement all abstract methods.

Member Access Modifiers

1. Methods and instance (nonlocal) variables are known as "members."
2. Members can use all four access levels: public, protected, default, private.
3. Member access comes in two forms:
4. Code in one class can access a member of another class.
5. A subclass can inherit a member of its superclass.

6. If a class cannot be accessed, its members cannot be accessed.
7. Determine class visibility before determining member visibility.
8. Public members can be accessed by all other classes, even in different packages.
9. If a superclass member is public, the subclass inherits it—regardless of package.
10. Members accessed without the dot operator (.) must belong to the same class.
11. this. always refers to the currently executing object.
12. this.aMethod() is the same as just invoking aMethod().
13. Private members can be accessed only by code in the same class.
14. Private members are not visible to subclasses, so private members cannot be inherited.
15. Default and protected members differ only in when subclasses are involved:
16. Default members can be accessed only by other classes in the same package.
17. Protected members can be accessed by other classes in the same package, plus subclasses regardless of package.
18. Protected = package plus kids (kids meaning subclasses).

19. For subclasses outside the package, the protected member can be accessed only through inheritance; a subclass outside the package cannot access a protected member by using a reference to an instance of the superclass (in other words, inheritance is the only mechanism for a subclass outside the package to access a protected member of its superclass).

20. A protected member inherited by a subclass from another package is, in practice, private to all other classes (in other words, no other classes from the subclass' package or any other package will have access to the protected member from the subclass).

Local Variables

1. Local (method, automatic, stack) variable declarations cannot have access modifiers.
2. final is the only modifier available to local variables.
3. Local variables don't get default values, so they must be initialized before use.

Other Modifiers—Members

1. Final methods cannot be overridden in a subclass.
2. Abstract methods have been declared, with a signature and return type, but have not been implemented.
3. Abstract methods end in a semicolon—no curly braces.
4. Three ways to spot a nonabstract method:
5. The method is not marked abstract.
6. The method has curly braces.
7. The method has code between the curly braces.
8. The first nonabstract (concrete) class to extend an abstract class must implement all of the abstract class' abstract methods.
9. Abstract methods must be implemented by a subclass, so they must be inheritable. For that reason:
10.Abstract methods cannot be private.
11.Abstract methods cannot be final.
12.The synchronized modifier applies only to methods. 13.Synchronized methods can have any access control and can also be marked final. 14.Synchronized methods cannot be abstract.
15.The native modifier applies only to methods.
16.The strictfp modifier applies only to classes and methods.

17.Instance variables can

a. Have any access control

b. Be marked final or transient

18.Instance variables cannot be declared abstract, synchronized, native, or strictfp.

19.It is legal to declare a local variable with the same name as an instance variable; this is called “shadowing.”

20. Final variables have the following properties:

a. Final variables cannot be reinitialized once assigned a value.

b. Final reference variables cannot refer to a different object once the object has been assigned to the final variable.

c. Final reference variables must be initialized before the constructor completes.

21. There is no such thing as a final object. An object reference marked final does not mean the object itself is immutable.

22. The transient modifier applies only to instance variables.

23. The volatile modifier applies only to instance variables.

Static variables and methods

1. They are not tied to any particular instance of a class.

2. An instance of a class does not need to exist in order to use static members of the class.

3. There is only one copy of a static variable per class and all instances share it.

4. Static variables get the same default values as instance variables.

5. A static method (such as main()) cannot access a nonstatic (instance) variable.

6. Static members are accessed using the class name: ClassName.theStaticMethodName()

7. Static members can also be accessed using an instance reference variable, someObj.theStaticMethodName() but that’s just a syntax trick; the static method won’t know anything about the instance referred to by the variable used to invoke the method. The compiler uses the class type of the reference variable to determine which static method to invoke.

8. Static methods cannot be overridden, although they can be redeclared/redefined by a subclass. So although static methods can sometimes appear to be overridden, polymorphism will not apply

Declaration Rules

1. A source code file can have only one public class.

2. If the source file contains a public class, the file name should match the public class name.

3. A file can have only one package statement, but can have multiple import statements.

4. The package statement (if any) must be the first line in a source file.

5. The import statements (if any) must come after the package and before the class declaration.

6. If there is no package statement, import statements must be the first statements in the source file.

7. Package and import statements apply to all classes in the file.

8. A file can have more than one nonpublic class.

9. Files with no public classes have no naming restrictions.

10. In a file, classes can be listed in any order (there is no forward referencing problem).

11. Import statements only provide a typing shortcut to a class’ fully qualified name.

12. Import statements cause no performance hits and do not increase the size of your code.

13. If you use a class from a different package, but do not import the class, you must use the fully qualified name of the class everywhere the class is used in code.

14. Import statements can coexist with fully qualified class names in a source file.

15. Imports ending in ‘.*;’ are importing all classes within a package.
16. Imports ending in ‘;’ are importing a single class.
17. You must use fully qualified names when you have different classes from different packages, with the same class name; an import statement will not be explicit enough.

Properties of main()

1. It must be marked static.
2. It must have a void return type.
3. It must have a single String array argument; the name of the argument is flexible, but the convention is args.
4. For the purposes of the exam, assume that the main() method must be public.
5. Improper main() method declarations (or the lack of a main() method) cause a runtime error, not a compiler error.
6. In the declaration of main(), the order of public and static can be switched, and args can be renamed.

7. Other overloaded methods named main() can exist legally in the class, but if none of them match the expected signature for the main() method, then the JVM won’t be able to use that class to start your application running.

java.lang.Runnable

1. You must memorize the java.lang.Runnable interface; it has a single
2. method you must implement: public void run {}.

Interface Implementation

1. Interfaces are contracts for what a class can do, but they say nothing about the way in which the class must do it.
2. Interfaces can be implemented by any class, from any inheritance tree.
3. An interface is like a 100-percent abstract class, and is implicitly abstract whether you type the abstract modifier in the declaration or not.
4. An interface can have only abstract methods, no concrete methods allowed.
5. Interfaces are by default public and abstract—explicit declaration of these modifiers is optional.
6. Interfaces can have constants, which are always implicitly public, static, and final.
7. Interface constant declarations of public, static, and final are optional in any combination.
8. A legal nonabstract implementing class has the following properties:
9. It provides concrete implementations for all methods from the interface.
10. It must follow all legal override rules for the methods it implements.
11. It must not declare any new checked exceptions for an implementation method.

12. It must not declare any checked exceptions that are broader than the exceptions declared in the interface method.
13. It may declare runtime exceptions on any interface method implementation regardless of the interface declaration.
14. It must maintain the exact signature and return type of the methods it implements (but does not have to declare the exceptions of the interface).
15. A class implementing an interface can itself be abstract.

16. An abstract implementing class does not have to implement the interface methods (but the first concrete subclass must).
17. A class can extend only one class (no multiple inheritance), but it can implement many.
18. Interfaces can extend one or more other interfaces.
19. Interfaces cannot extend a class, or implement a class or interface.
20. When taking the exam, verify that interface and class declarations are legal before verifying other code logic.

Java Operators

1. The result of performing most operations is either a boolean or a numeric value.
2. Variables are just bit holders with a designated type.
3. A reference variable's bits represent a way to get to an object.
4. An unassigned reference variable's bits represent null.
5. There are 12 assignment operators: =, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=.
6. Numeric expressions always result in at least an int-sized result─never smaller.
7. Floating-point numbers are implicitly doubles (64 bits).
8. Narrowing a primitive truncates the high-order bits.
9. Two' s complement means: flip all the bits, then add 1.
10. Compound assignments (e.g. +=) perform an automatic cast.

Reference Variables

1. When creating a new object, e.g., Button b = new Button();, three things happen:
a. Make a reference variable named b, of type Button
b. Create a new Button object
c. Refer the reference variable b to the Button object
2. Reference variables can refer to subclasses of the declared type but not superclasses.

String Objects and References

1. String objects are immutable, cannot be changed.

2. When you use a String reference variable to modify a String:

a. A new string is created (the old string is immutable).
b. The reference variable refers to the new string.

Comparison Operators

1. Comparison operators always result in a boolean value (true or false).
2. There are four comparison operators: >, >=, <, <=.
3. When comparing characters, Java uses the ASCII or Unicode value of the number as the numerical value.

instanceof Operator

1. instanceof is for reference variables only, and checks for whether this object is of a particular type.
2. The instanceof operator can be used only to test objects (or null) against class types that are in the same class hierarchy.

3. For interfaces, an object is “of a type” if any of its superclasses implement the interface in question.

Equality Operators

1. Four types of things can be tested: numbers, characters, booleans, reference variables.
2. There are two equality operators: == and !=.

Arithmetic Operators

1. There are four primary operators: add, subtract, multiply, and divide.
2. The remainder operator returns the remainder of a division.
3. When floating-point numbers are divided by zero, they return positive or negative infinity, except when the dividend is also zero, in which case you get NaN.
4. When the remainder operator performs a floating-point divide by zero, it will not cause a runtime exception.
5. When integers are divided by zero, a runtime ArithmeticException is thrown.
6. When the remainder operator performs an integer divide by zero, a runtime ArithmeticException is thrown.

String Concatenation Operator

1. If either operand is a String, the + operator concatenates the operands.
2. If both operands are numeric, the + operator adds the operands.

Increment/Decrement Operators

1. Prefix operator runs before the value is used in the expression.
2. Postfix operator runs after the value is used in the expression.
3. In any expression, both operands are fully evaluated before the operator is applied.
4. Final variables cannot be incremented or decremented.

Shift Operators

1. There are three shift operators: >>, <<, >>>; the first two are signed, the last is unsigned.
2. Shift operators can only be used on integer types.
3. Shift operators can work on all bases of integers (octal, decimal, or hexadecimal).
4. Except for the unusual cases of shifting an int by a multiple of 32 or a long by a multiple of 64 (these shifts result in no change to the original values), bits are filled as follows:

a. << fills the right bits with zeros.
b. >> fills the left bits with whatever value the original sign bit (leftmost bit) held.

c. >>> fills the left bits with zeros (negative numbers will become positive).

5. All bit shift operands are promoted to at least an int.
6. For int shifts > 32 or long shifts > 64, the actual shift value is the remainder of the right operand / divided by 32 or 64, respectively.

Bitwise Operators

1. There are three bitwise operators—&, ^, |—and a bitwise complement, operator ~.
2. The & operator sets a bit to 1 if both operand's bits are set to 1. 3. The ^ operator sets a bit to 1 if exactly one operand's bit is set to 1. 4. The | operator sets a bit to 1 if at least one operand's bit is set to 1.
5. The ~ operator reverses the value of every bit in the single operand.

Ternary (Conditional Operator)

1. Returns one of two values based on whether a boolean expression is true or false.
a. The value after the ? is the ‘if true return’.
b. The value after the : is the ‘if false return’.

Casting

1. Implicit casting (you write no code) happens when a widening conversion occurs.
2. Explicit casting (you write the cast) is required when a narrowing conversion is desired.
3. Casting a floating point to an integer type causes all digits to the right of the decimal point to be lost (truncated).
4. Narrowing conversions can cause loss of data—the most significant bits (leftmost) can be lost.

Logical Operators

1. There are four logical operators: &, |, &&, ||.
2. Logical operators work with two expressions that must resolve to boolean values.
3. The && and & operators return true only if both operands are true.
4. The || and | operators return true if either or both operands are true.
5. The && and || operators are known as short-circuit operators.
6. The && operator does not evaluate the right operand if the left operand is false.
7. The || does not evaluate the right operand if the left operand is true.
8. The & and | operators always evaluate both operands.

Passing Variables into Methods

1. Methods can take primitives and/or object references as arguments.
2. Method arguments are always copies—of either primitive variables or reference variables.
3. Method arguments are never actual objects (they can be references to objects).
4. In practice, a primitive argument is a completely detached copy of the original primitive.
5. In practice, a reference argument is another copy of a reference to the original object.

Writing Code Using if and switch Statements

1. The if statement must have all expressions enclosed by at least one pair of parentheses.
2. The only legal argument to an if statement is a boolean, so the if test can be only on an expression that resolves to a boolean or a boolean variable.
3. Watch out for boolean assignments (=) that can be mistaken for Boolean equality (==) tests:
4. boolean x = false;
5. if (x = true) { } // an assignment, so x will always be true!

6. Curly braces are optional for if blocks that have only one conditional statement. But watch out for misleading indentations.

7. Switch statements can evaluate only the byte, short, int, and char data types. You can't say long s = 30;

switch(s) { }

8. The case argument must be a literal or final variable! You cannot have a case that includes a nonfinal variable, or a range of values.

9. If the condition in a switch statement matches a case value, execution will run through all code in the switch following the matching case statement until a break or the end of the switch statement is encountered. In other words, the matching case is just the entry point into the case block, but unless there's a break statement, the matching case is not the only case code that runs.

10. The default keyword should be used in a switch statement if you want to execute some code when none of the case values match the conditional value. 11. The default block can be located anywhere in the switch block, so if no case matches, the default block will be entered, and if the default does not contain a break, then code will continue to execute (fall-through) to the end of the switch or until the break statement is encountered.

Writing Code Using Loops

1. A for statement does not require any arguments in the declaration, but has three parts: declaration and/or initialization, boolean evaluation, and the iteration expression.

2. If a variable is incremented or evaluated within a for loop, it must be declared before the loop, or within for loop declaration.

3. A variable declared (not just initialized) within the for loop declaration cannot be accessed outside the for loop (in other words, code below the for loop won't be able to use the variable).

4. You can initialize more than one variable in the first part of the for loop declaration; each variable initialization must be separated by a comma.

5. You cannot use a number (old C-style language construct) or anything that does not evaluate to a boolean value as a condition for an if statement or looping construct. You can't, for example, say:

if (x) unless x is a boolean variable.

6. The do-while loop will enter the body of the loop at least once, even if the test condition is not met.

Using break and continue

1. An unlabeled break statement will cause the current iteration of the innermost looping construct to stop and the next line of code following the loop to be executed.
2. An unlabeled continue statement will cause the current iteration of the innermost loop to stop, and the condition of that loop to be checked, and if the condition is met, perform the loop again.

3. If the break statement or the continue statement is labeled, it will cause similar action to occur on the labeled loop, not the innermost loop.
4. If a continue statement is used in a for loop, the iteration statement is executed, and the condition is checked again.

Catching an Exception Using try and catch

1. Exceptions come in two flavors: checked and unchecked.
2. Checked exceptions include all subtypes of Exception, excluding classes that extend RuntimeException.
3. Checked exceptions are subject to the handle or declare rule; any method that might throw a checked exception (including methods that invoke methods that can throw a checked exception) must either declare the exception using the throws keyword, or handle the exception with an appropriate try/catch.
4. Subtypes of Error or RuntimeException are unchecked, so the compiler doesn't enforce the handle or declare rule. You're free to handle them, and you're free to declare them, but the compiler doesn't care one way or the other.
5. If you use an optional finally block, it will always be invoked, regardless of whether an exception in the corresponding try is thrown or not, and regardless of whether a thrown exception is caught or not.
6. The only exception to the finally-will-always-be-called rule is that a finally will not be invoked if the JVM shuts down. That could happen if code from the try or catch blocks calls System.exit(), in which case the JVM will not start your finally block.
7. Just because finally is invoked does not mean it will complete.
8. Code in the finally block could itself raise an exception or issue a System.exit().
9. Uncaught exceptions propagate back through the call stack, starting from the method where the exception is thrown and ending with either the first method that has a corresponding catch for that exception type or a JVM shutdown (which happens if the exception gets to main(), and main() is "ducking" the exception by declaring it).
10. You can create your own exceptions, normally by extending Exception or one of its subtypes. Your exception will then be considered a checked exception, and the compiler will enforce the handle or declare rule for that exception.
11. All catch blocks must be ordered from most specific to most general.

12. For example, if you have a catch clause for both IOException and Exception, you must put the catch for IOException first (in order, top to bottom in your code). Otherwise, the IOException would be caught by catch(Exception e), because a catch argument can catch the specified exception or any of its subtypes! The compiler will stop you from defining catch clauses that can never be reached (because it sees that the more specific exception will be caught first by the more general catch).

Working with the Assertion Mechanism

1. Assertions give you a way to test your assumptions during development and debugging.
2. Assertions are typically enabled during testing but disabled during deployment.
3. You can use assert as a keyword (as of version 1.4) or an identifier, but not both together. To compile older code that uses assert as an identifier (for example, a method name), use the -source 1.3 command-line flag to javac.

4. Assertions are disabled at runtime by default. To enable them, use a command-line flag -ea or -enableassertions.

5. You can selectively disable assertions using the -da or -disableassertions flag.

6. If you enable or disable assertions using the flag without any arguments, you're enabling or disabling assertions in general. You can combine enabling and disabling switches to have assertions enabled for some classes and/or packages, but not others.

7. You can enable or disable assertions in the system classes with the -esa or -dsa flags.

8. You can enable and disable assertions on a class-by-class basis, using the following syntax: java -ea -da:MyClass TestClass

9. You can enable and disable assertions on a package basis, and any package you specify also includes any subpackages (packages further down the directory hierarchy).

10. Do not use assertions to validate arguments to public methods.

11. Do not use assert expressions that cause side effects. Assertions aren't guaranteed to always run, so you don't want behavior that changes depending on whether assertions are enabled.

12. Do use assertions—even in public methods—to validate that a particular code block will never be reached. You can use assert false; for code that should never be reached, so that an assertion error is thrown immediately if the assertstatement is executed.

13. Do not use assert expressions that can cause side effects.

Encapsulation, IS-A, HAS-A

1. The goal of encapsulation is to hide implementation behind an interface (or API).

2. Encapsulated code has two features:

a. Instance variables are kept protected (usually with the private modifier).

b. Getter and setter methods provide access to instance variables.

3. IS-A refers to inheritance.

4. IS-A is expressed with the keyword extends.

5. “IS-A,” “inherits from,” “is derived from,” and “is a subtype of” are all equivalent expressions.

6. HAS-A means an instance of one class “has a” reference to an instance of another class.

Overriding and Overloading

1. Methods can be overridden or overloaded; constructors can be overloaded but not overridden.

2. Abstract methods must be overridden by the first concrete (nonabstract) subclass.

3. With respect to the method it overrides, the overriding method a. Must have the same argument list

b. Must have the same return type

c. Must not have a more restrictive access modifier

d. May have a less restrictive access modifier

e. Must not throw new or broader checked exceptions

f. May throw fewer or narrower checked exceptions, or any unchecked exception

4. Final methods cannot be overridden.

5. Only inherited methods may be overridden.

6. A subclass uses super.overriddenMethodName to call the superclass version of an overridden method.

7. Overloading means reusing the same method name, but with different arguments.
8. Overloaded methods
a. Must have different argument lists
b. May have different return types, as long as the argument lists are also different
c. May have different access modifiers
d. May throw different exceptions
9. Methods from a superclass can be overloaded in a subclass.
10. Polymorphism applies to overriding, not to overloading
11. Object type determines which overridden method is used at runtime.
12. Reference type determines which overloaded method will be used at compile time.

Instantiation and Constructors

1. Objects are constructed:

a. You cannot create a new object without invoking a constructor.

b. Each superclass in an object's inheritance tree will have a constructor called.

2. Every class, even abstract classes, has at least one constructor.

3. Constructors must have the same name as the class.

4. Constructors do not have a return type. If there is a return type, then it is simply a method with the same name as the class, and not a constructor.

5. Constructor execution occurs as follows:

a. The constructor calls its superclass constructor, which calls its superclass constructor, and so on all the way up to the Object constructor.

b. The Object constructor executes and then returns to the calling constructor, which runs to completion and then returns to its calling constructor, and so on back down to the completion of the constructor of the actual instance being created.

6. Constructors can use any access modifier (even private!).
7. The compiler will create a default constructor if you don' t create any constructors in your class.
8. The default constructor is a no-arg constructor with a no-arg call to super().
9. The first statement of every constructor must be a call to either this() (an overloaded constructor) or super().
10. The compiler will add a call to super() if you do not, unless you have already put in a call to this().
11. Instance methods and variables are only accessible after the super constructor runs.
12. Abstract classes have constructors that are called when a concrete subclass is instantiated.
13. Interfaces do not have constructors.
14. If your superclass does not have a no-arg constructor, you must create a constructor and insert a call to super() with arguments matching those of the superclass constructor
. 15. Constructors are never inherited, thus they cannot be overridden.
16. A constructor can be directly invoked only by another constructor (using a call to super() or this()).
17. Issues with calls to this():

a. May appear only as the first statement in a constructor.
b. The argument list determines which overloaded constructor is called.
c. Constructors can call constructors can call constructors, and so on, but sooner or later one of them better call super() or the stack will explode.
d. this() and super() cannot be in the same constructor. You can have one or the other, but never both.

Return Types

1. Overloaded methods can change return types; overridden methods cannot.
2. Object reference return types can accept null as a return value.
3. An array is a legal return type, both to declare and return as a value.
4. For methods with primitive return types, any value that can be implicitly converted to the return type can be returned.
5. Nothing can be returned from a void, but you can return nothing. You’re allowed to simply say return, in any method with a void return type, to bust out of a method early. But you can’t return nothing from a method with a non-void return type.
6. For methods with an object reference return type, a subclass of that type can be returned.
7. For methods with an interface return type, any implementer of that interface can be returned.

Using the java.lang.String Class

1. String objects are immutable, and String reference variables are not.
2. If you create a new String without assigning it, it will be lost to your program.
3. If you redirect a String reference to a new String, the old String can be lost.
4. String methods use zero-based indexes, except for the second argument of substring().
5. The String class is final—its methods can’t be overridden.
6. When a String literal is encountered by the VM, it is added to the pool.
7. Strings have a method named length(), arrays have an attribute named length.
8. StringBuffers are mutable–they can change without creating a new object.
9. StringBuffer methods act on the invoking object, but objects can change without an explicit assignment in the statement.
10. StringBuffer equals() is not overridden; it doesn’t compare values.
11. In all sections, remember that chained methods are evaluated from left to right.

Using the java.lang.Math Class

1. The abs() method is overloaded to take an int, a long, a float, or a double.
2. The abs() method can return a negative if the argument is the minimum int or long value equal to the value of Integer.MIN_VALUE or Long.MIN_VALUE, respectively.
3. The max() method is overloaded to take int, long, float, or double arguments.
4. The min() method is overloaded to take int, long, float, or double arguments.
5. The random() method returns a double greater than or equal to 0.0 and less than 1.0.
6. The random() does not take any arguments.
7. The methods ceil(), floor(), and round() all return integer equivalent floating-point numbers, ceil() and floor() return doubles,
8. round() returns an int if it was passed a float, or it returns a long if it was passed a double.

9. The round() method is overloaded to take a float or a double.
10. The methods sin(), cos(), and tan() take a double angle in radians.
11. The method sqrt() can return NaN if the argument is NaN or less than zero.
12. Floating-point numbers can be divided by 0.0 without error; the result is either positive or negative infinity.
13. NaN is not equal to anything, not even itself.

Using Wrappers

1. The wrapper classes correlate to the primitive types.
2. Wrappers have two main functions:
a. To wrap primitives so that they can be handled like objects
b. To provide utility methods for primitives (usually conversions)
3. Other than Character and Integer, wrapper class names are the primitive's name, capitalized.
4. Wrapper constructors can take a String or a primitive, except for Character, which can only take a char.
5. A Boolean object can't be used like a boolean primitive.
6. The three most important method families are
a. xxxValue() Takes no arguments, returns a primitive
b. parseXxx() Takes a String, returns a primitive, is static, throws NFE
c. valueOf() Takes a String, returns a wrapped object, is static, throws NFE
7. Radix refers to bases (typically) other than 10; binary is radix 2, octal = 8, hex = 16.

Using equals()

1. Use == to compare primitive variables.
2. Use == to determine if two reference variables refer to the same object.
3. == compares bit patterns, either primitive bits or reference bits.
4. Use equals() to determine if two objects are meaningfully equivalent.
5. The String and Wrapper classes override equals() to check for values.
6. The StringBuffer class equals() is not overridden; it uses == under the covers.
7. The compiler will not allow == if the classes are not in the same hierarchy.
8. Wrappers won' t pass equals() if they are in different classes.

Overriding hashCode() and equals()

1. The critical methods in class Object are equals(), finalize(), hashCode(), and toString().
2. equals(), hashCode(), and toString() are public (finalize() is protected).
3. Fun facts about toString():
a. Override toString() so that System.out.println() or other methods can see something useful.
b. Override toString() to return the essence of your object's state.
4. Use == to determine if two reference variables refer to the same object.
5. Use equals() to determine if two objects are meaningfully equivalent.
6. If you don' t override equals(), your objects won' t be useful hashtable/hashmap keys.
7. If you don' t override equals(), two different objects can' t be considered the same.
8. Strings and wrappers override equals() and make good hashtable/hashmap keys.

9. When overriding equals(), use the instanceof operator to be sure you’re evaluating an appropriate class.
10. When overriding equals(), compare the objects’ significant attributes.
11. Highlights of the equals() contract:
a. Reflexive: x.equals(x) is true.
b. Symmetric: If x.equals(y) is true, then y.equals(x) must be true.
c. Transitive: If x.equals(y) is true, and y.equals(z) is true, then z.equals(x) is true.
d. Consistent: Multiple calls to x.equals(y) will return the same result.
e. Null: If x is not null, then x.equals(null) is false.
12. If x.equals(y) is true, then x.hashCode() == y.hashCode() must be true.
13. If you override equals(), override hashCode().
14. Classes HashMap, Hashtable, LinkedHashMap, and LinkedHashSet use hashing.
15. A legal hashCode() override compiles and runs.
16. An appropriate hashCode() override sticks to the contract
. 17. An efficient hashCode() override distributes keys randomly across a wide range of buckets.
18. To reiterate: if two objects are equal, their hashcodes must be equal.
19. It’s legal for a hashCode() method to return the same value for all instances (although in practice it’s very inefficient).
20. Highlights of the hashCode() contract:
a. Consistent: Multiple calls to x.hashCode() return the same integer.
b. If x.equals(y) is true, then x.hashCode() == y.hashCode() must be true.
c. If x.equals(y) is false, then x.hashCode() == y.hashCode() can be either true or false, but false will tend to create better efficiency.
21. Transient variables aren’t appropriate for equals() and hashCode().

Collections

1. Common collection activities include adding objects, removing objects, verifying object inclusion, retrieving objects, and iterating.
2. Three meanings for “collection”:
a. collection—Represents the data structure in which objects are stored
b. Collection—java.util.Collection—Interface from which Set and List extend c. Collections—A class that holds static collection utility methods
3. Three basic flavors of collections include Lists, Sets, Maps:
a. Lists of things: Ordered, duplicates allowed, with an index
b. Sets of things: May or may not be ordered and/or sorted, duplicates not allowed
c. Maps of things with keys: May or may not be ordered and/or sorted, duplicate keys not allowed
4. Four basic subflavors of collections include Sorted, Unsorted, Ordered, Unordered.
5. Ordered means iterating through a collection in a specific, nonrandom order.
6. Sorted means iterating through a collection in a natural sorted order.
7. Natural means alphabetic, numeric, or programmer-defined, whichever applies.
8. Key attributes of common collection classes:
a. ArrayList: Fast iteration and fast random access
b. Vector: Like a somewhat slower ArrayList, mainly due to its synchronized methods
c. LinkedList: Good for adding elements to the ends, i.e., stacks and queues

d. HashSet: Assures no duplicates, provides no ordering
e. LinkedHashSet: No duplicates; iterates by insertion order or last accessed (new with 1.4)
f. TreeSet: No duplicates; iterates in natural sorted order
g. HashMap: Fastest updates (key/value pairs); allows one null key, many null values
h. Hashtable: Like a slower HashMap (as with Vector, due to its synchronized methods). No null values or null keys allowed
i. LinkedHashMap: Faster iterations; iterates by insertion order or last accessed, allows one null key, many null values (new with 1.4)

j. TreeMap: A sorted map, in natural order

Garbage Collection

1. In Java, garbage collection provides some automated memory management.
2. All objects in Java live on the heap.
3. The heap is also known as the garbage collectible heap.
4. The purpose of garbage collecting is to find and delete objects that can’t be reached.
5. Only the JVM decides exactly when to run the garbage collector.
6. You (the programmer) can only recommend when to run the garbage collector.
7. You can’t know the G.C. algorithm; maybe it uses mark and sweep, maybe it’s generational and/or iterative.
8. Objects must be considered eligible before they can be garbage collected.
9. An object is eligible when no live thread can reach it.
10. To reach an object, a live thread must have a live, reachable reference variable to that object.
11. Java applications can run out of memory.
12. Islands of objects can be garbage collected, even though they have references.
13. To reiterate: garbage collection can’t be forced.
14. Request garbage collection with System.gc(); (recommended).
15. Class Object has a finalize() method.
16. The finalize() method is guaranteed to run once and only once before the garbage collector deletes an object.
17. Since the garbage collector makes no guarantees, finalize()may never run.
18. You can uneligibilize an object from within finalize().

Inner Classes

1. A “regular” inner class is declared inside the curly braces of another class, but outside any method or other code block.
2. An inner class is a full-fledged member of the enclosing (outer) class, so it can be marked with an access modifier as well as the abstract or final modifiers (but of course, never both abstract and final together—remember that abstract means it must be subclassed, whereas final means it cannot be subclassed).
3. An inner class instance shares a special relationship with an instance of the enclosing class. This relationship gives the inner class access to all of the outer class’ members, including those marked private.

4. To instantiate an inner class, you must have a reference to an instance of the outer class.

5. From code within the enclosing class, you can instantiate the inner class using only the name of the inner class, as follows:

```
MyInner mi = new MyInner();
```

6. From code outside the enclosing class' instance methods, you can instantiate the inner class only by using both the inner and outer class names, and a reference to the outer class as follows:

```
MyOuter mo = new MyOuter();

MyOuter.MyInner inner = mo.new MyInner();
```

7. From code within the inner class, the keyword this holds a reference to the inner class instance. To reference the outer this (in other words, the instance of the outer class that this inner instance is tied to) precede the keyword this with the outer class name as follows: MyOuter.this;

Method-Local Inner Classes

1. A method-local inner class is defined within a method of the enclosing class.
2. For the inner class to be used, you must instantiate it, and that instantiation must happen within the same method, but after the class definition code.
3. A method-local inner class cannot use variables declared within the method (including parameters) unless those variables are marked final.
4. The only modifiers you can apply to a method-local inner class are abstract and final. (Never both at the same time, though.)

Anonymous Inner Classes

1. Anonymous inner classes have no name, and their type must be either a subclass of the named type or an implementer of the named interface.
2. An anonymous inner class is always created as part of a statement, so don't forget to close the statement after the class definition, with a curly brace. This is one of the rare times you'll see a curly brace followed by a semicolon in Java.
3. Because of polymorphism, the only methods you can call on an anonymous inner class reference are those defined in the reference variable class (or interface), even though the anonymous class is really a subclass or implementer of the reference variable type.
4. An anonymous inner class can extend one subclass, or implement one interface. Unlike nonanonymous classes (inner or otherwise), an anonymous inner class cannot do both. In other words, it cannot both extend a class and implement an interface, nor can it implement more than one interface.
5. An argument-local inner class is declared, defined, and automatically instantiated as part of a method invocation. The key to remember is that the class is being defined within a method argument, so the syntax will end the class definition with a curly brace, followed by a closing parenthesis to end the method call, followed by a semicolon to end the statement: });

Static Nested Classes

1. Static nested classes are inner classes marked with the static modifier.
2. Technically, a static nested class is not an inner class, but instead is considered a top-level nested class.
3. Because the nested class is static, it does not share any special relationship with an instance of the outer class. In fact, you don’t need an instance of the outer class to instantiate a static nested class.
4. Instantiating a static nested class requires using both the outer and nested class names as follows:

```
BigOuter.Nested n = new BigOuter.Nested();
```

5. A static nested class cannot access nonstatic members of the outer class, since it does not have an implicit reference to any outer instance (in other words, the nested class instance does not get an outer this reference).